Abjad API

Release 2.12

Trevor Bača, Josiah Oberholtzer, Víctor Adán

March 25, 2013

CONTENTS

Part I

Core composition packages

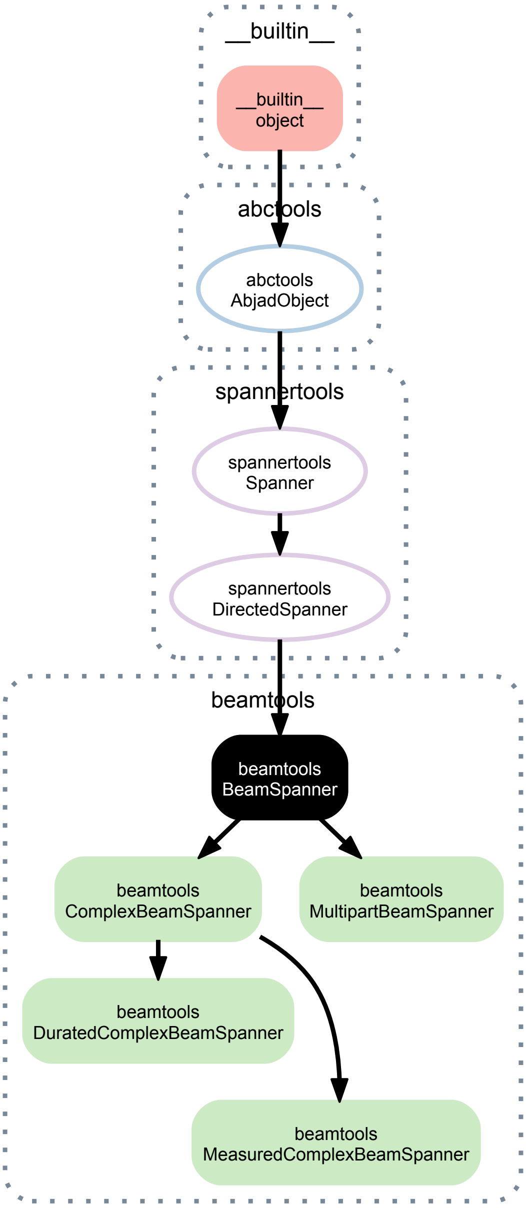
CHAPTER

ONE

BEAMTOOLS

1.1 Concrete Classes

1.1.1 beamtools.BeamSpanner



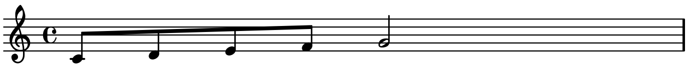
class beamtools.**BeamSpanner** (*components=None*, *direction=None*)

Abjad beam spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'2")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
    g'2
}
```

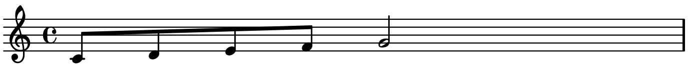
```
>>> show(staff)
```



```
>>> beamtools.BeamSpanner(staff[:4])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
    g'2
}
```

```
>>> show(staff)
```



Return beam spanner.

Read-only properties

BeamSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

BeamSpanner.**duration**

Sum of prolated duration of all components in spanner.

BeamSpanner.**duration_in_seconds**

Sum of duration of all leaves in spanner, in seconds.

BeamSpanner.**leaves**

Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

BeamSpanner.**override**

LilyPond grob override component plug-in.

`BeamSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`BeamSpanner.`**`set`**
LilyPond context setting component plug-in.

`BeamSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`BeamSpanner.`**`timespan`**
Read-only timespan of spanner.

`BeamSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

`BeamSpanner.`**`direction`**

Methods

`BeamSpanner.`**`append`**`(`*component*`)`
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`BeamSpanner.`**`append_left`**`(`*component*`)`
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`BeamSpanner.`**`clear`**`()`
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`BeamSpanner.`**`extend`**`(`*components*`)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c’8, d’8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c’8, d’8, e’8, f’8)
```

Return none.

`BeamSpanner.`**`extend_left`**`(`*components*`)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e’8, f’8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c’8, d’8, e’8, f’8)
```

Return none.

`BeamSpanner.`**`fracture`**`(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

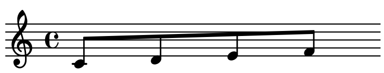
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c’8, d’8, e’8, f’8)
```

```
>>> f(voice)
\new Voice {
    c’8 [
    d’8
    e’8
    f’8 ]
}
```

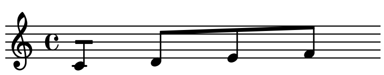
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c’8, d’8, e’8, f’8), BeamSpanner(c’8), BeamSpanner(d’8, e’8, f’8))
```

```
>>> f(voice)
\new Voice {
    c’8 [ ]
    d’8 [
    e’8
    f’8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`BeamSpanner.fuse(spanner)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`BeamSpanner.index(component)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`BeamSpanner.pop()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

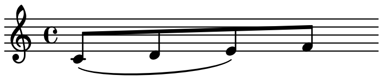
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

BeamSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

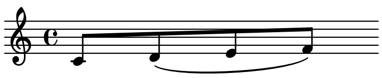
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

BeamSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

BeamSpanner.**__contains__**(*expr*)

BeamSpanner.**__copy__**(**args*)

BeamSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

BeamSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BeamSpanner.**__getitem__**(*expr*)

BeamSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BeamSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BeamSpanner.**__len__**()

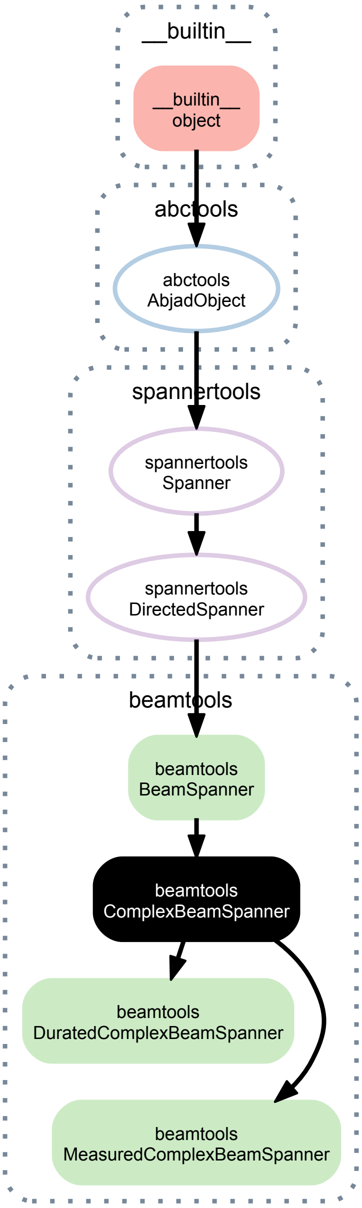
BeamSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

BeamSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

BeamSpanner.**__repr__**()

1.1.2 beamtools.ComplexBeamSpanner



class beamtools.**ComplexBeamSpanner**(*components=None*, *lone=False*, *direction=None*)

Abjad complex beam spanner:

```
>>> staff = Staff("c'16 e'16 r16 f'16 g'2")
```

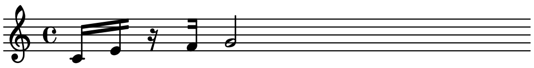
```
>>> f(staff)
\new Staff {
    c'16
    e'16
    r16
    f'16
    g'2
}
```

```
>>> show(staff)
```

```
>>> beamtools.ComplexBeamSpanner(staff[:4])
ComplexBeamSpanner(c'16, e'16, r16, f'16)
```

```
>>> f(staff)
\new Staff {
    \set stemLeftBeamCount = #0
    \set stemRightBeamCount = #2
    c'16 [
    \set stemLeftBeamCount = #2
    \set stemRightBeamCount = #2
    e'16 ]
    r16
    \set stemLeftBeamCount = #2
    \set stemRightBeamCount = #0
    f'16 [ ]
    g'2
}
```

```
>>> show(staff)
```



Return complex beam spanner.

Read-only properties

`ComplexBeamSpanner.`**`components`**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`ComplexBeamSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`ComplexBeamSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`ComplexBeamSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`ComplexBeamSpanner.`**`override`**
LilyPond grob override component plug-in.

`ComplexBeamSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`ComplexBeamSpanner.`**`set`**
LilyPond context setting component plug-in.

`ComplexBeamSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

ComplexBeamSpanner.**timespan**
Read-only timespan of spanner.

ComplexBeamSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

ComplexBeamSpanner.**direction**

ComplexBeamSpanner.**lone**
Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='left')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #0
c'16 [ ]
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='right')
```

```
>>> f(note)
\set stemLeftBeamCount = #0
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='both')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=True)
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=False)
```

```
>>> f(note)
c'16
```

Set to `'left'`, `'right'`, `'both'`, true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

Methods

`ComplexBeamSpanner.append(component)`

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`ComplexBeamSpanner.append_left(component)`

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`ComplexBeamSpanner.clear()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`ComplexBeamSpanner.extend(components)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`ComplexBeamSpanner.extend_left(components)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`ComplexBeamSpanner.`**`fracture`**`(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

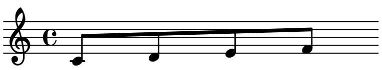
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

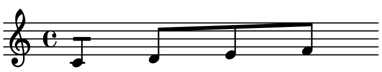
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`ComplexBeamSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
```

```
        d'8
        e'8
        f'8 ]
    }
```

Return list.

`ComplexBeamSpanner.index(component)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`ComplexBeamSpanner.pop()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

`ComplexBeamSpanner.pop_left()`

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

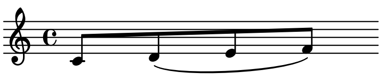
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

ComplexBeamSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

ComplexBeamSpanner.**__contains__**(*expr*)

ComplexBeamSpanner.**__copy__**(**args*)

ComplexBeamSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ComplexBeamSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ComplexBeamSpanner.**__getitem__**(*expr*)

ComplexBeamSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ComplexBeamSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ComplexBeamSpanner.**__len__**()

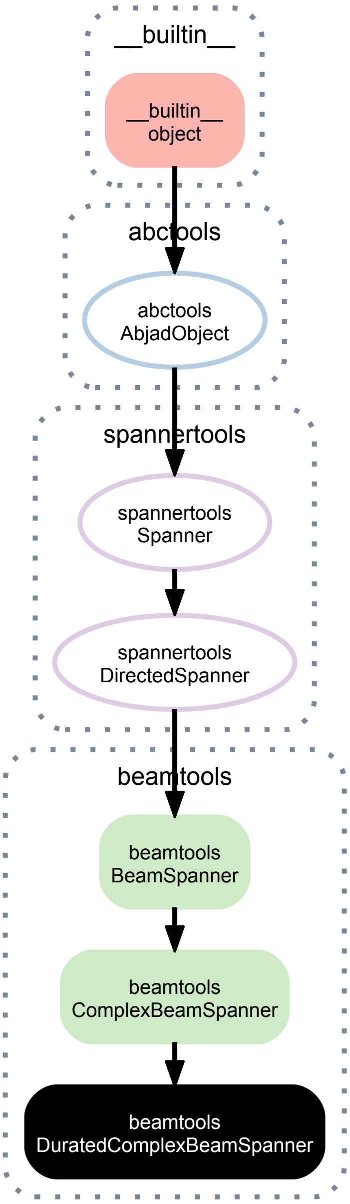
ComplexBeamSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

ComplexBeamSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ComplexBeamSpanner.**__repr__**()

1.1.3 beamtools.DuratedComplexBeamSpanner



class `beamtools.DuratedComplexBeamSpanner` (*components=None*, *durations=None*, *span=1*, *lone=False*, *direction=None*)

Abjad durated complex beam spanner:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
```

```
>>> show(staff)
```

```
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = beamtools.DuratedComplexBeamSpanner(staff[:], durations, 1)
```

```
>>> f(staff)
\new Staff {
    \set stemLeftBeamCount = #0
    \set stemRightBeamCount = #2
    c'16 [
    \set stemLeftBeamCount = #2
```

```
        \set stemRightBeamCount = #1
        d'16
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #2
        e'16
        \set stemLeftBeamCount = #2
        \set stemRightBeamCount = #0
        f'16 ]
    }
```

```
>>> show(staff)
```

Beam all beamable leaves in spanner explicitly.

Group leaves in spanner according to *durations*.

Span leaves between duration groups according to *span*.

Return durated complex beam spanner.

Read-only properties

DuratedComplexBeamSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

DuratedComplexBeamSpanner.**duration**
Sum of prolated duration of all components in spanner.

DuratedComplexBeamSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

DuratedComplexBeamSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

DuratedComplexBeamSpanner.**override**
LilyPond grob override component plug-in.

DuratedComplexBeamSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

DuratedComplexBeamSpanner.**set**
LilyPond context setting component plug-in.

DuratedComplexBeamSpanner.**storage_format**
Storage format of Abjad object.

Return string.

DuratedComplexBeamSpanner.**timespan**
Read-only timespan of spanner.

DuratedComplexBeamSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

DuratedComplexBeamSpanner.**direction**

DuratedComplexBeamSpanner.**durations**
Get spanner leaf group durations:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = beamtools.DuratedComplexBeamSpanner(staff[:], durations)
>>> beam.durations
[Duration(1, 8), Duration(1, 8)]
```

Set spanner leaf group durations:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = beamtools.DuratedComplexBeamSpanner(staff[:], durations)
>>> beam.durations = [Duration(1, 4)]
>>> beam.durations
[Duration(1, 4)]
```

Set iterable.

DuratedComplexBeamSpanner.**lone**
Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='left')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #0
c'16 [ ]
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='right')
```

```
>>> f(note)
\set stemLeftBeamCount = #0
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='both')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=True)
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=False)
```

```
>>> f(note)
c'16
```

Set to `'left'`, `'right'`, `'both'`, true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

DuratedComplexBeamSpanner.**span**

Get top-level beam count:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = beamtools.DuratedComplexBeamSpanner(staff[:], durations, 1)
>>> beam.span
1
```

Set top-level beam count:

```
>>> staff = Staff("c'16 d'16 e'16 f'16")
>>> durations = [Duration(1, 8), Duration(1, 8)]
>>> beam = beamtools.DuratedComplexBeamSpanner(staff[:], durations, 1)
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

Methods

DuratedComplexBeamSpanner.**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

DuratedComplexBeamSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`DuratedComplexBeamSpanner.`**`clear`**`()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`DuratedComplexBeamSpanner.`**`extend`**`(`*components*`)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DuratedComplexBeamSpanner.`**`extend_left`**`(`*components*`)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DuratedComplexBeamSpanner.`**`fracture`**`(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

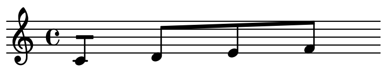
```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`DuratedComplexBeamSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`DuratedComplexBeamSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`DuratedComplexBeamSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

`DuratedComplexBeamSpanner.`**`pop_left`**`()`

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

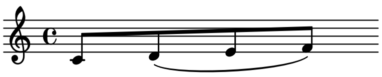
```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
```

```
        f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

DuratedComplexBeamSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

DuratedComplexBeamSpanner.**__contains__**(*expr*)

DuratedComplexBeamSpanner.**__copy__**(**args*)

DuratedComplexBeamSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DuratedComplexBeamSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DuratedComplexBeamSpanner.**__getitem__**(*expr*)

DuratedComplexBeamSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DuratedComplexBeamSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DuratedComplexBeamSpanner.**__len__**()

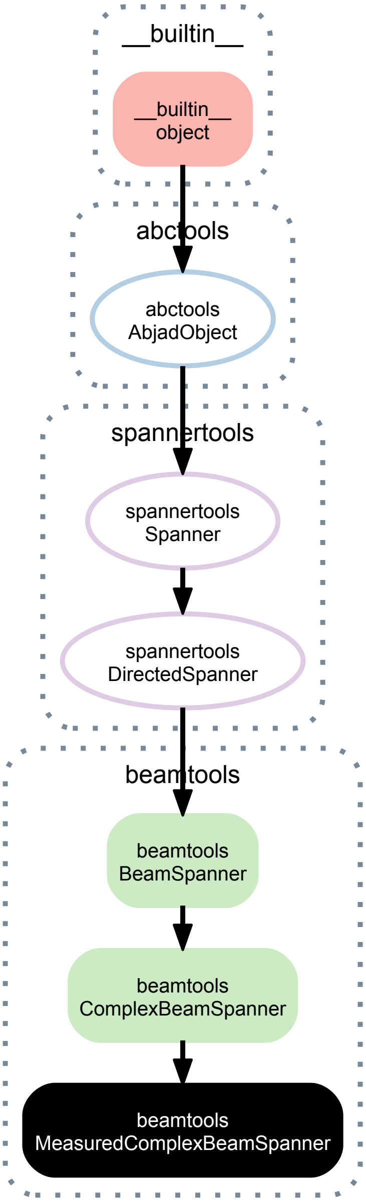
DuratedComplexBeamSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

DuratedComplexBeamSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DuratedComplexBeamSpanner.**__repr__**()

1.1.4 beamtools.MeasuredComplexBeamSpanner



class beamtools.**MeasuredComplexBeamSpanner**(*components=None*, *lone=False*, *span=1*, *direction=None*)

Abjad measured complex beam spanner:

```
>>> staff = Staff([Measure((2, 16), "c'16 d'16"), Measure((2, 16), "e'16 f'16")])
```

```
>>> show(staff)
```

```
>>> beamtools.MeasuredComplexBeamSpanner(staff.leaves)
MeasuredComplexBeamSpanner(c'16, d'16, e'16, f'16)
```

```
>>> f(staff)
\new Staff {
    {
```

```
            \time 2/16
            \set stemLeftBeamCount = #0
            \set stemRightBeamCount = #2
            c'16 [
            \set stemLeftBeamCount = #2
            \set stemRightBeamCount = #1
            d'16
        }
        {
            \set stemLeftBeamCount = #1
            \set stemRightBeamCount = #2
            e'16
            \set stemLeftBeamCount = #2
            \set stemRightBeamCount = #0
            f'16 ]
        }
    }
```

```
>>> show(staff)
```

Beam leaves in spanner explicitly.

Group leaves by measures.

Format top-level *span* beam between measures.

Return measured complex beam spanner.

Read-only properties

`MeasuredComplexBeamSpanner.`**`components`**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`MeasuredComplexBeamSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`MeasuredComplexBeamSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`MeasuredComplexBeamSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`MeasuredComplexBeamSpanner.`**`override`**
LilyPond grob override component plug-in.

`MeasuredComplexBeamSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`MeasuredComplexBeamSpanner.`**`set`**
LilyPond context setting component plug-in.

MeasuredComplexBeamSpanner.**storage_format**
Storage format of Abjad object.

Return string.

MeasuredComplexBeamSpanner.**timespan**
Read-only timespan of spanner.

MeasuredComplexBeamSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

MeasuredComplexBeamSpanner.**direction**

MeasuredComplexBeamSpanner.**lone**
Beam lone leaf and force beam nibs to left:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='left')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #0
c'16 [ ]
```

Beam lone leaf and force beam nibs to right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='right')
```

```
>>> f(note)
\set stemLeftBeamCount = #0
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and force beam nibs to both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone='both')
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Beam lone leaf and accept LilyPond default nibs at both left and right:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=True)
```

```
>>> f(note)
\set stemLeftBeamCount = #2
\set stemRightBeamCount = #2
c'16 [ ]
```

Do not beam lone leaf:

```
>>> note = Note("c'16")
```

```
>>> beam = beamtools.ComplexBeamSpanner([note], lone=False)
```

```
>>> f(note)
c'16
```

Set to `'left'`, `'right'`, `'both'`, true or false as shown above.

Ignore this setting when spanner contains more than one leaf.

`MeasuredComplexBeamSpanner.`**span**

Get top-level beam count:

```
>>> staff = Staff([Measure((2, 16), "c'16 d'16"), Measure((2, 16), "e'16 f'16")])
>>> beam = beamtools.MeasuredComplexBeamSpanner(staff.leaves)
>>> beam.span
1
```

Set top-level beam count:

```
>>> staff = Staff([Measure((2, 16), "c'16 d'16"), Measure((2, 16), "e'16 f'16")])
>>> beam = beamtools.MeasuredComplexBeamSpanner(staff.leaves)
>>> beam.span = 2
>>> beam.span
2
```

Set nonnegative integer.

Methods

`MeasuredComplexBeamSpanner.`**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`MeasuredComplexBeamSpanner.`**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`MeasuredComplexBeamSpanner.`**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`MeasuredComplexBeamSpanner.`**`extend`**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`MeasuredComplexBeamSpanner.`**`extend_left`**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`MeasuredComplexBeamSpanner.`**`fracture`**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

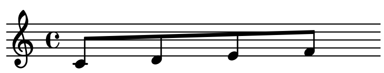
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

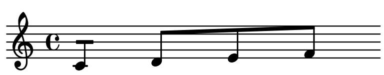
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

MeasuredComplexBeamSpanner.**fuse**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c’8 [
    d’8 ]
    e’8 [
    f’8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c’8, d’8), BeamSpanner(e’8, f’8), BeamSpanner(c’8, d’8, e’8, f’8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c’8 [
    d’8
    e’8
    f’8 ]
}
```

Return list.

MeasuredComplexBeamSpanner.**index**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e’8, f’8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

MeasuredComplexBeamSpanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c’8 d’8 e’8 f’8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c’8, d’8, e’8, f’8)
```

```
>>> f(voice)
\new Voice {
    c’8 (
    d’8
    e’8
    f’8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f’8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

MeasuredComplexBeamSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

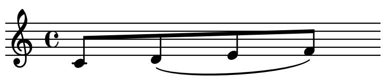
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

MeasuredComplexBeamSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

MeasuredComplexBeamSpanner.**__contains__**(*expr*)

MeasuredComplexBeamSpanner.**__copy__**(**args*)

MeasuredComplexBeamSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

MeasuredComplexBeamSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MeasuredComplexBeamSpanner.**__getitem__**(*expr*)

MeasuredComplexBeamSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MeasuredComplexBeamSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MeasuredComplexBeamSpanner.**__len__**()

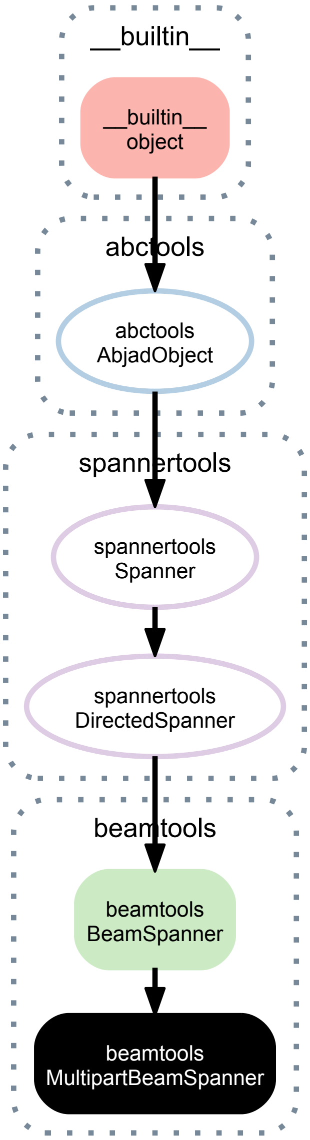
MeasuredComplexBeamSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

MeasuredComplexBeamSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MeasuredComplexBeamSpanner.**__repr__**()

1.1.5 beamtools.MultipartBeamSpanner

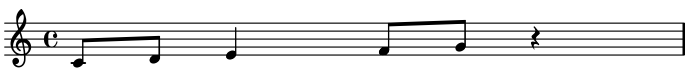


class `beamtools.`**`MultipartBeamSpanner`**(*components=None*, *direction=None*)

New in version 2.0. Abjad multipart beam spanner:

```
>>> staff = Staff("c'8 d'8 e'4 f'8 g'8 r4")
```

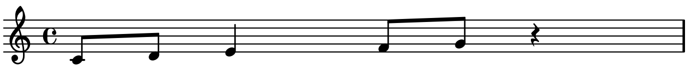
```
>>> show(staff)
```



```
>>> beamtools.MultipartBeamSpanner(staff[:])
MultipartBeamSpanner(c'8, d'8, e'4, f'8, g'8, r4)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
    e'4
    f'8 [
    g'8 ]
    r4
}
```

```
>>> show(staff)
```



Avoid rests.

Avoid large-duration notes.

Return multipart beam spanner.

Read-only properties

`MultipartBeamSpanner.`**`components`**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`MultipartBeamSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`MultipartBeamSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`MultipartBeamSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`MultipartBeamSpanner.`**`override`**
LilyPond grob override component plug-in.

`MultipartBeamSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`MultipartBeamSpanner.`**`set`**
LilyPond context setting component plug-in.

`MultipartBeamSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`MultipartBeamSpanner.`**`timespan`**
Read-only timespan of spanner.

`MultipartBeamSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

`MultipartBeamSpanner.`**`direction`**

Methods

`MultipartBeamSpanner.`**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`MultipartBeamSpanner.`**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`MultipartBeamSpanner.`**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`MultipartBeamSpanner.`**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`MultipartBeamSpanner.`**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`MultipartBeamSpanner.`**`fracture`**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

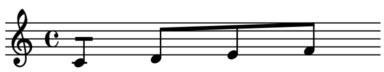
```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`MultipartBeamSpanner.`**`fuse`**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
```

```
        d'8
        e'8
        f'8 ]
    }
```

Return list.

MultipartBeamSpanner.**index**(*component*)
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

MultipartBeamSpanner.**pop**()
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

MultipartBeamSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

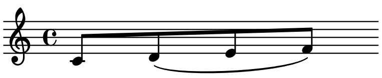
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

`MultipartBeamSpanner.`**`__call__`**`(`*expr*`)`
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

`MultipartBeamSpanner.`**`__contains__`**`(`*expr*`)`

`MultipartBeamSpanner.`**`__copy__`**`(`**args*`)`

`MultipartBeamSpanner.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`MultipartBeamSpanner.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`MultipartBeamSpanner.__getitem__(expr)`

`MultipartBeamSpanner.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`MultipartBeamSpanner.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`MultipartBeamSpanner.__len__()`

`MultipartBeamSpanner.__lt__(expr)`
Trivial comparison to allow doctests to work.

`MultipartBeamSpanner.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`MultipartBeamSpanner.__repr__()`

1.2 Functions

1.2.1 beamtools.apply_beam_spanners_to_measures_in_expr

`beamtools.apply_beam_spanners_to_measures_in_expr(expr)`
New in version 1.1. Apply beam spanners to measures in *expr*:

```
>>> staff = Staff(r"abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
}
```

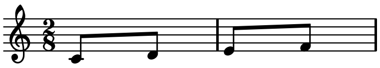
```
>>> show(staff)
```

```
>>> beamtools.apply_beam_spanners_to_measures_in_expr(staff)
[BeamSpanner(|2/8(2)|), BeamSpanner(|2/8(2)|)]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [
        d'8 ]
```

```
        }
        {
            e'8 [
            f'8 ]
        }
    }
```

```
>>> show(staff)
```



Return list of beams created.

1.2.2 beamtools.apply_complex_beam_spanners_to_measures_in_expr

beamtools.**apply_complex_beam_spanners_to_measures_in_expr**(*expr*)

New in version 2.0. Apply complex beam spanners to measures in *expr*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
}
```

```
>>> show(staff)
```

```
>>> beamtools.apply_complex_beam_spanners_to_measures_in_expr(staff)
[ComplexBeamSpanner(|2/8(2)|), ComplexBeamSpanner(|2/8(2)|)]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        \set stemLeftBeamCount = #0
        \set stemRightBeamCount = #1
        c'8 [
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #0
        d'8 ]
    }
    {
        \set stemLeftBeamCount = #0
        \set stemRightBeamCount = #1
        e'8 [
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #0
        f'8 ]
    }
}
```

```
>>> show(staff)
```

Return list of beams created.

1.2.3 beamtools.apply_durated_complex_beam_spanner_to_measures

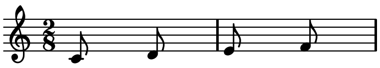
beamtools.**apply_durated_complex_beam_spanner_to_measures**(*measures*)

New in version 1.1. Apply durated complex beam spanner to *measures*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
}
```

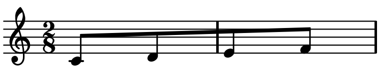
```
>>> show(staff)
```



```
>>> measures = staff[:]
>>> beamtools.apply_durated_complex_beam_spanner_to_measures(measures)
DuratedComplexBeamSpanner(|2/8(2)|, |2/8(2)|)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        \set stemLeftBeamCount = #0
        \set stemRightBeamCount = #1
        c'8 [
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #1
        d'8
    }
    {
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #1
        e'8
        \set stemLeftBeamCount = #1
        \set stemRightBeamCount = #0
        f'8 ]
    }
}
```

```
>>> show(staff)
```



Set beam spanner durations to preprolated measure durations.

Return beam spanner created.

1.2.4 beamtools.apply_multipart_beam_spanner_to_bottommost_tuplets_in_expr

beamtools.**apply_multipart_beam_spanner_to_bottommost_tuplets_in_expr**(*expr*)

Beam bottommost tuplets in *expr*:

```
>>> staff = Staff(3 * Tuplet(Fraction(2, 3), "c'8 d'8 e'8"))
```

```
f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    \times 2/3 {
        c'8
        d'8
        e'8
    }
}
```

```
>>> show(staff)
```


```
>>> beamtools.apply_multipart_beam_spanner_to_bottommost_tuplets_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
    }
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
    }
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
    }
}
```

```
>>> show(staff)
```

Return none.

1.2.5 beamtools.get_beam_spanner_attached_to_component

beamtools.**get_beam_spanner_attached_to_component**(*component*)

New in version 2.0. Get the only beam spanner attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(staff)
```

```
>>> spanner = beamtools.get_beam_spanner_attached_to_component(staff[0])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner is beam
True
```

Return beam spanner.

Raise missing spanner error when no beam spanner attached to *component*.

Raise extra spanner error when more than one beam spanner attached to *component*.

1.2.6 beamtools.is_beamable_component

beamtools.**is_beamable_component**(*expr*)
New in version 1.1. True when *expr* is a beamable component. Otherwise false:

```
>>> beamtools.is_beamable_component(Note(13, (1, 16)))
True
```

Return boolean.

1.2.7 beamtools.is_component_with_beam_spanner_attached

beamtools.**is_component_with_beam_spanner_attached**(*expr*)
New in version 2.0. True when *expr* is component with beam spanner attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
```

```
>>> beamtools.is_component_with_beam_spanner_attached(staff[0])
True
```

Otherwise false:

```
>>> note = Note("c'8")
```

```
>>> beamtools.is_component_with_beam_spanner_attached(note)
False
```

Return boolean.

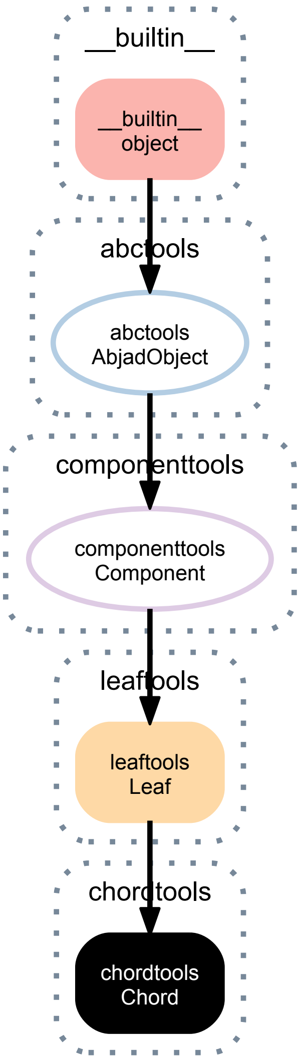
CHAPTER

TWO

CHORDTOOLS

2.1 Concrete Classes

2.1.1 chordtools.Chord



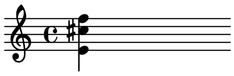
class chordtools.**Chord**(**args*, ***kwargs*)

Abjad model of a chord:

```
>>> chord = Chord([4, 13, 17], (1, 4))
```

```
>>> chord
Chord("<e' cs'' f''>4")
```

```
>>> show(chord)
```



Return Chord instance.

Read-only properties

`Chord.`**`descendants`**
Read-only reference to component descendants score selection.

`Chord.`**`duration`**

`Chord.`**`duration_in_seconds`**

`Chord.`**`fingered_pitches`**
Read-only fingered pitches:

```
>>> staff = Staff("<c''' e'''>4 <d''' fs'''>4")
>>> glockenspiel = instrumenttools.Glockenspiel()(staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Glockenspiel }
    \set Staff.shortInstrumentName = \markup { Gkspl. }
    <c' e'>4
    <d' fs'>4
}
```

```
>>> staff[0].fingered_pitches
(NamedChromaticPitch("c'"), NamedChromaticPitch("e'"))
```

Return tuple of named chromatic pitches.

`Chord.`**`leaf_index`**

`Chord.`**`lilypond_format`**

`Chord.`**`lineage`**
Read-only reference to component lineage score selection.

`Chord.`**`multiplied_duration`**

`Chord.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Chord.`**`parent`**

`Chord.`**`parentage`**
Read-only reference to component parentage score selection.

`Chord.`**`preprolated_duration`**

`Chord.`**`prolation`**

`Chord.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Chord.`**`sounding_pitches`**
Read-only sounding pitches:

```
>>> staff = Staff("<c''' e'''>4 <d''' fs'''>4")
>>> glockenspiel = instrumenttools.Glockenspiel()(staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Glockenspiel }
    \set Staff.shortInstrumentName = \markup { Gkspl. }
    <c' e'>4
    <d' fs'>4
}
```

```
>>> staff[0].sounding_pitches
(NamedChromaticPitch("c'''"), NamedChromaticPitch("e'''"))
```

Return tuple of named chromatic pitches.

Chord.**spanners**
Read-only reference to unordered set of spanners attached to component.

Chord.**storage_format**
Storage format of Abjad object.

Return string.

Chord.**timespan**
Read-only timespan of component.

Chord.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

Chord.**duration_multiplier**

Chord.**note_heads**
Get read-only tuple of note heads in chord:

```
>>> chord = Chord([7, 12, 16], (1, 4))
>>> chord.note_heads
(NoteHead("g'"), NoteHead("c''"), NoteHead("e''"))
```

Set chord note heads from any iterable:

```
>>> chord = Chord([7, 12, 16], (1, 4))
>>> chord.note_heads = [0, 2, 6]
>>> chord
Chord("<c' d' fs'>4")
```

Chord.**written_duration**

Chord.**written_pitch_indication_is_at_sounding_pitch**

Chord.**written_pitch_indication_is_nonsemantic**

Chord.**written_pitches**
Get read-only tuple of pitches in chord:

```
>>> chord = Chord([7, 12, 16], (1, 4))
>>> chord.written_pitches
(NamedChromaticPitch("g'"), NamedChromaticPitch("c''"), NamedChromaticPitch("e''"))
```

Set chord pitches from any iterable:

```
>>> chord = Chord([7, 12, 16], (1, 4))
>>> chord.written_pitches = [0, 2, 6]
>>> chord
Chord("<c' d' fs'>4")
```

Methods

`Chord.`**`append`**(*note_head*)

Append *note_head* to chord:

```
>>> chord = Chord([4, 13, 17], (1, 4))
>>> chord
Chord("<e' cs'' f''>4")
```

```
>>> chord.append(19)
>>> chord
Chord("<e' cs'' f'' g''>4")
```

Sort chord note heads automatically after append and return none.

`Chord.`**`extend`**(*note_heads*)

Extend chord with *note_heads*:

```
>>> chord = Chord([4, 13, 17], (1, 4))
>>> chord
Chord("<e' cs'' f''>4")
```

```
>>> chord.extend([2, 12, 18])
>>> chord
Chord("<d' e' c'' cs'' f'' fs''>4")
```

Sort chord note heads automatically after extend and return none.

`Chord.`**`pop`**(*i=-1*)

Remove note head at index *i* in chord:

```
>>> chord = Chord([4, 13, 17], (1, 4))
>>> chord
Chord("<e' cs'' f''>4")
```

```
>>> chord.pop(1)
NoteHead("cs''")
```

```
>>> chord
Chord("<e' f''>4")
```

Return note head.

`Chord.`**`remove`**(*note_head*)

Remove *note_head* from chord:

```
>>> chord = Chord([4, 13, 17], (1, 4))
>>> chord
Chord("<e' cs'' f''>4")
```

```
>>> chord.remove(chord[1])
>>> chord
Chord("<e' f''>4")
```

Return none.

Special methods

`Chord.`**`__and__`**(*arg*)

`Chord.`**`__contains__`**(*arg*)

`Chord.`**`__copy__`**(**args*)

`Chord.`**`__delitem__`**(*i*)

Chord.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Chord.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Chord.**__getitem__**(*i*)

Chord.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Chord.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Chord.**__len__**()

Chord.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Chord.**__mul__**(*n*)

Chord.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Chord.**__or__**(*arg*)

Chord.**__repr__**()

Chord.**__rmul__**(*n*)

Chord.**__setitem__**(*i*, *arg*)

Chord.**__str__**()

Chord.**__sub__**(*arg*)

Chord.**__xor__**(*arg*)

2.2 Functions

2.2.1 chordtools.all_are_chords

chordtools.**all_are_chords**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad chords:

```
>>> chords = [Chord("<c' e' g'>4"), Chord("<c' f' a'>4")]
```

```
>>> chordtools.all_are_chords(chords)
True
```

True when *expr* is an empty sequence:

```
>>> chordtools.all_are_chords([])
True
```

Otherwise false:

```
>>> chordtools.all_are_chords('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

2.2.2 chordtools.arpeggiate_chord

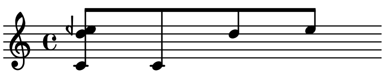
`chordtools.arpeggiate_chord(chord)`

New in version 1.1. Arpeggiate *chord*:

```
>>> chord = Chord("<c' d'' eqf''>8")
```

```
>>> arpeggio = chordtools.arpeggiate_chord(chord)
>>> arpeggio
[Note("c'8"), Note("d''8"), Note("eqf''8")]
```

```
>>> staff = Staff([chord])
>>> staff.extend(arpeggio)
>>> show(staff)
```



Arpeggiated notes inherit *chord* written duration.

Arpeggiated notes do not inherit other *chord* attributes.

Return list of newly constructed notes.

2.2.3 chordtools.change_defective_chord_to_note_or_rest

`chordtools.change_defective_chord_to_note_or_rest(chord)`

New in version 1.1. Change zero-length *chord* to rest:

```
>>> chord = Chord([], (3, 16))
```

```
>>> chord
Chord('<>8.')
```

```
>>> chordtools.change_defective_chord_to_note_or_rest(chord)
Rest('r8.')
```

Change length-one chord to note:

```
>>> chord = Chord("<cs''>8.")
```

```
>>> chord
Chord("<cs''>8.")
```

```
>>> chordtools.change_defective_chord_to_note_or_rest(chord)
Note("cs''8.")
```

Return chords with length greater than one unchanged:

```
>>> chord = Chord("<c' c'' cs''>8.")
```

```
>>> chord
Chord("<c' c'' cs''>8.")
```

```
>>> chordtools.change_defective_chord_to_note_or_rest(chord)
Chord("<c' c'' cs''>8.")
```

Return notes unchanged:

```
>>> note = Note("c'4")
```

```
>>> note
Note("c'4")
```

```
>>> chordtools.change_defective_chord_to_note_or_rest(note)
Note("c'4")
```

Return rests unchanged:

```
>>> rest = Rest('r4')
```

```
>>> rest
Rest('r4')
```

```
>>> chordtools.change_defective_chord_to_note_or_rest(rest)
Rest('r4')
```

Return note, rest, chord or none.

2.2.4 chordtools.divide_chord_by_chromatic_pitch_number

chordtools.**divide_chord_by_chromatic_pitch_number**(*chord*, *pitch=NamedChromaticPitch('b')*)

New in version 1.1. Divide *chord* by chromatic *pitch* number:

```
>>> chord = Chord(range(12), Duration(1, 4))
```

```
>>> chord
Chord("<c' cs' d' ef' e' f' fs' g' af' a' bf' b'>4")
```

```
>>> chordtools.divide_chord_by_chromatic_pitch_number(chord, pitchtools.NamedChromaticPitch(6))
(Chord("<fs' g' af' a' bf' b'>4"), Chord("<c' cs' d' ef' e' f'>4"))
```

Input *chord* may be a note, rest or chord but not a skip.

Zero-length parts return rests, length-one parts return notes and other parts return chords.

Return pair of newly constructed leaves.

2.2.5 chordtools.divide_chord_by_diatonic_pitch_number

chordtools.**divide_chord_by_diatonic_pitch_number**(*chord*, *pitch=NamedChromaticPitch('b')*)

New in version 1.1. Divide *chord* by diatonic *pitch* number:

```
>>> chord = Chord(range(12), Duration(1, 4))
```

```
>>> chord
Chord("<c' cs' d' ef' e' f' fs' g' af' a' bf' b'>4")
```

```
>>> chordtools.divide_chord_by_diatonic_pitch_number(chord, pitchtools.NamedChromaticPitch(6))
(Chord("<f' fs' g' af' a' bf' b'>4"), Chord("<c' cs' d' ef' e'>4"))
```

Input *chord* may be a note, rest or chord but not a skip.

Zero-length parts return as rests, length-one parts return as notes and other parts return as chords.

Return pair of newly constructed leaves.

2.2.6 chordtools.get_arithmetic_mean_of_chord

chordtools.**get_arithmetic_mean_of_chord**(*chord*)

New in version 2.0. Get arithmetic mean of chromatic pitch number of pitches in *chord*:

```
>>> chord = Chord("<g' c'' e''>4")
```

```
>>> chordtools.get_arithmetic_mean_of_chord(chord)
11.666666666666666
```

Return none when *chord* is empty:

```
>>> chord = Chord("< >4")
```

```
>>> chordtools.get_arithmetic_mean_of_chord(chord) is None
True
```

Return number or none.

2.2.7 chordtools.get_note_head_from_chord_by_pitch

chordtools.**get_note_head_from_chord_by_pitch**(*chord*, *pitch*)

New in version 2.0. Get note head from *chord* by *pitch*:

```
>>> chord = Chord("<c'' d'' b''>4")
```

```
>>> chordtools.get_note_head_from_chord_by_pitch(chord, 14)
NoteHead("d''")
```

Raise missing note head error when *chord* contains no note head with pitch equal to *pitch*.

Raise extra note head error when *chord* contains more than one note head with pitch equal to *pitch*.

2.2.8 chordtools.make_tied_chord

chordtools.**make_tied_chord**(*pitches*, *duration*, *decrease_durations_monotonically=True*, *forbidden_written_duration=None*)

Returns a list of chords to fill the given duration.

Chords returned are tie spanned.

2.2.9 chordtools.yield_all_subchords_of_chord

chordtools.**yield_all_subchords_of_chord**(*chord*)

New in version 2.0. Yield all subchords of *chord* in binary string order:

```
>>> chord = Chord("<c' d' af' a'>4")
```

```
>>> for subchord in chordtools.yield_all_subchords_of_chord(chord):
...     subchord
...
Rest('r4')
Note("c'4")
Note("d'4")
Chord("<c' d'>4")
Note("af'4")
Chord("<c' af'>4")
Chord("<d' af'>4")
Chord("<c' d' af'>4")
Note("a'4")
Chord("<c' a'>4")
Chord("<d' a'>4")
Chord("<c' d' a'>4")
Chord("<af' a'>4")
```

```
Chord("<c' af' a'>4")
Chord("<d' af' a'>4")
Chord("<c' d' af' a'>4")
```

Include empty chord as rest.

Return generator of newly constructed leaves.

2.2.10 chordtools.yield_groups_of_chords_in_sequence

chordtools.**yield_groups_of_chords_in_sequence**(*sequence*)

New in version 2.0. Yield groups of chords in *sequence*:

```
>>> staff = Staff("c'8 d'8 r8 r8 <e' g'>8 <f' a'>8 g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    r8
    r8
    <e' g'>8
    <f' a'>8
    g'8
    a'8
    r8
    r8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for chord in chordtools.yield_groups_of_chords_in_sequence(staff):
...     chord
...
(Chord("<e' g'>8"), Chord("<f' a'>8"))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Return generator.

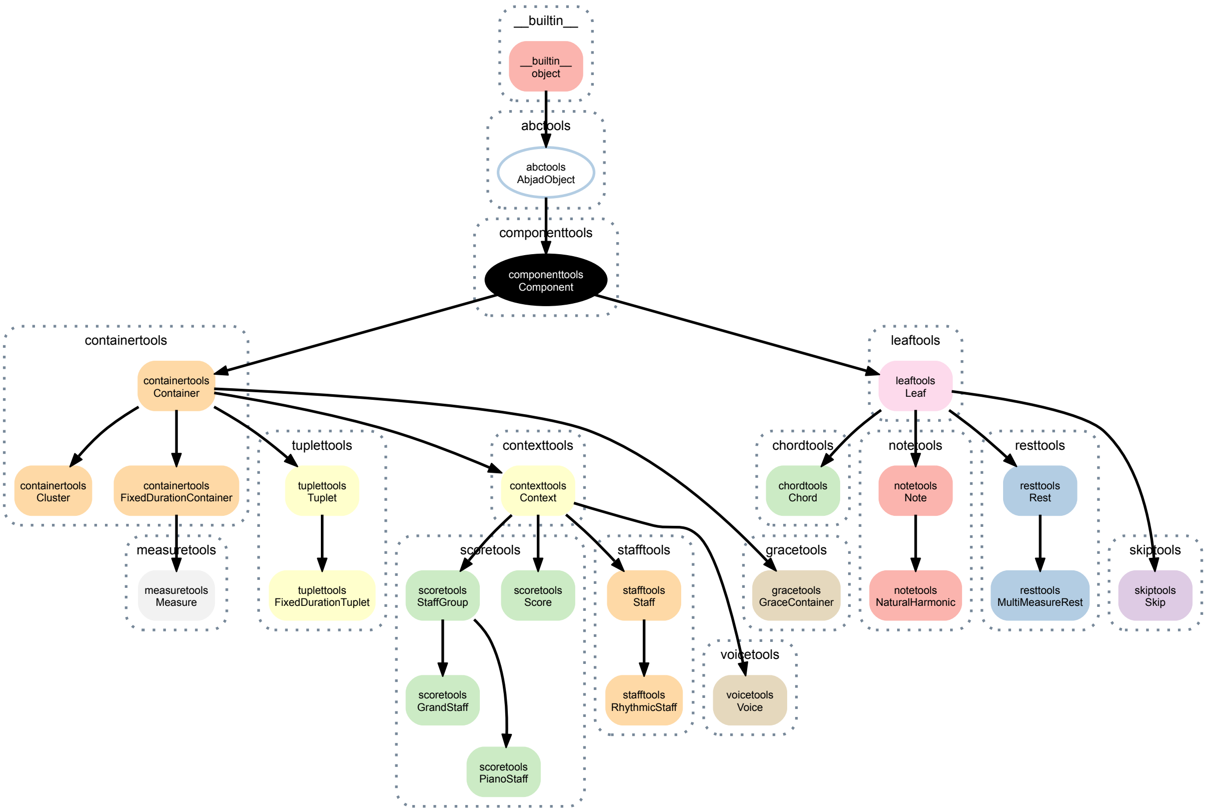
CHAPTER

THREE

COMPONENTTOOLS

3.1 Abstract Classes

3.1.1 componenttools.Component



class `componenttools.`**`Component`**

Read-only properties

`Component.`**`descendants`**
Read-only reference to component descendants score selection.

`Component.`**`duration`**

`Component.`**`duration_in_seconds`**

`Component.`**`lilypond_format`**

`Component.`**`lineage`**
Read-only reference to component lineage score selection.

`Component.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Component.`**`parent`**

`Component.`**`parentage`**
Read-only reference to component parentage score selection.

`Component.`**`prolation`**

`Component.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Component.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`Component.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Component.`**`timespan`**
Read-only timespan of component.

`Component.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Special methods

`Component.`**`__copy__`**(**args*)

`Component.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`Component.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Component.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`Component.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Component.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Component.`**`__mul__`**(*n*)

`Component.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

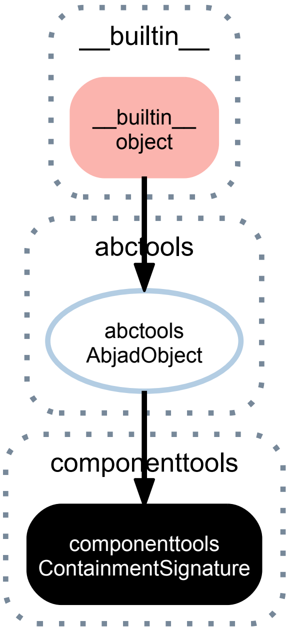
`Component.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

`Component.`**`__rmul__`**(*n*)

3.2 Concrete Classes

3.2.1 componenttools.ContainmentSignature



class componenttools.**ContainmentSignature**

New in version 2.9. Containment signature of Abjad component:

```
>>> score = Score(r"""\context Staff = "CustomStaff" { """
...     r"""\context Voice = "CustomVoice" { c' d' e' f' } }""")
>>> score.name = 'CustomScore'
```

```
>>> f(score)
\context Score = "CustomScore" <<
    \context Staff = "CustomStaff" {
        \context Voice = "CustomVoice" {
            c'4
            d'4
            e'4
            f'4
        }
    }
>>
```

```
>>> score.leaves[0].parentage.containment_signature
ContainmentSignature(Note-..., Voice-'CustomVoice', Staff-..., Score-'CustomScore')
```

Used for thread iteration behind the scenes.

Read-only properties

ContainmentSignature.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

ContainmentSignature.**__eq__**(*arg*)

ContainmentSignature.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContainmentSignature.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ContainmentSignature.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContainmentSignature.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

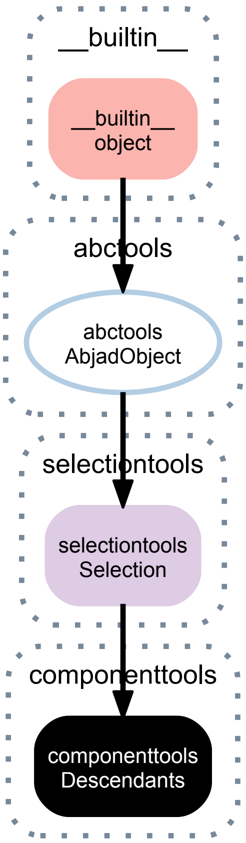
Raise exception.

ContainmentSignature.**__ne__**(*arg*)

ContainmentSignature.**__repr__**()

ContainmentSignature.**__str__**()

3.2.2 componenttools.Descendants



class componenttools.**Descendants**(*component*)
Abjad model of Component descendants:

```
>>> score = Score()
>>> score.append(Staff(r"""\new Voice = "Treble Voice" { c'4 }""",
...     name='Treble Staff'))
>>> score.append(Staff(r"""\new Voice = "Bass Voice" { b,4 }""",
...     name='Bass Staff'))
```

```
>>> f(score)
\new Score <<
    \context Staff = "Treble Staff" {
        \context Voice = "Treble Voice" {
            c'4
        }
    }
    \context Staff = "Bass Staff" {
        \context Voice = "Bass Voice" {
            b,4
        }
    }
>>
```

```
>>> for x in componenttools.Descendants(score): x
...
Score<<2>>
Staff-"Treble Staff"{1}
Voice-"Treble Voice"{1}
Note("c'4")
Staff-"Bass Staff"{1}
Voice-"Bass Voice"{1}
Note('b,4')
```

```
>>> for x in componenttools.Descendants(score['Bass Voice']): x
...
Voice-"Bass Voice"{1}
Note('b,4')
```

Descendants is treated as the selection of the component's improper descendants.

Return Descendants instance.

Read-only properties

`Descendants.`**component**
The component from which the selection was derived.

`Descendants.`**music**
Read-only tuple of components in selection.

`Descendants.`**storage_format**
Storage format of Abjad object.

Return string.

`Descendants.`**timespan**
Read-only timespan of selection.

Special methods

`Descendants.`**__add__**(*expr*)

`Descendants.`**__contains__**(*expr*)

`Descendants.`**__eq__**(*expr*)

`Descendants.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Descendants.`**__getitem__**(*expr*)

`Descendants.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Descendants.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Descendants.**__len__**()

Descendants.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

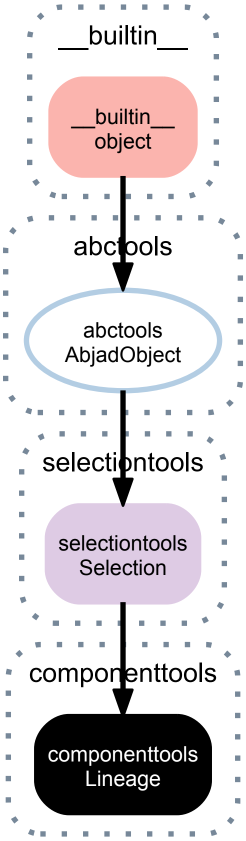
Raise exception.

Descendants.**__ne__**(*expr*)

Descendants.**__radd__**(*expr*)

Descendants.**__repr__**()

3.2.3 componenttools.Lineage



class componenttools.**Lineage**(*component*)
Abjad model of Component lineage:

```
>>> score = Score()
>>> score.append(Staff(r"""\new Voice = "Treble Voice" { c'4 }""",
... name='Treble Staff'))
>>> score.append(Staff(r"""\new Voice = "Bass Voice" { b,4 }""",
... name='Bass Staff'))
```

```
>>> f(score)
\new Score <<
    \context Staff = "Treble Staff" {
        \context Voice = "Treble Voice" {
            c'4
        }
    }
```

```
        \context Staff = "Bass Staff" {
            \context Voice = "Bass Voice" {
                b,4
            }
        }
    >>
```

```
>>> for x in componenttools.Lineage(score): x
...
Score<<2>>
Staff-"Treble Staff"{1}
Voice-"Treble Voice"{1}
Note("c'4")
Staff-"Bass Staff"{1}
Voice-"Bass Voice"{1}
Note('b,4')
```

```
>>> for x in componenttools.Lineage(score['Bass Voice']): x
...
Score<<2>>
Staff-"Bass Staff"{1}
Voice-"Bass Voice"{1}
Note('b,4')
```

Return Lineage instance.

Read-only properties

`Lineage.`**`component`**
The component from which the selection was derived.

`Lineage.`**`music`**
Read-only tuple of components in selection.

`Lineage.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Lineage.`**`timespan`**
Read-only timespan of selection.

Special methods

`Lineage.`**`__add__`**`(`*expr*`)`

`Lineage.`**`__contains__`**`(`*expr*`)`

`Lineage.`**`__eq__`**`(`*expr*`)`

`Lineage.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Lineage.`**`__getitem__`**`(`*expr*`)`

`Lineage.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Lineage.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Lineage.`**`__len__`**`()`

Lineage.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

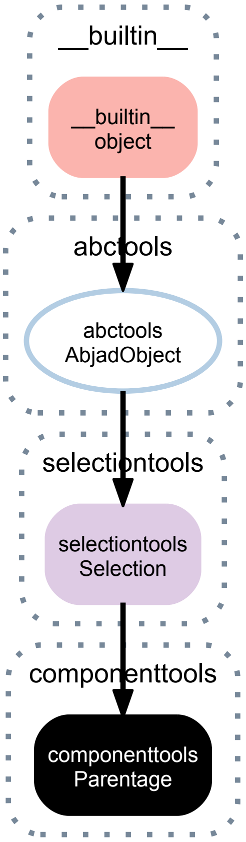
Raise exception.

Lineage.**__ne__**(*expr*)

Lineage.**__radd__**(*expr*)

Lineage.**__repr__**()

3.2.4 componenttools.Parentage



class componenttools.**Parentage**(*component*)
Abjad model of component parentage:

```
>>> score = Score()
>>> score.append(Staff(r"""\new Voice = "Treble Voice" { c'4 }""",
...     name='Treble Staff'))
>>> score.append(Staff(r"""\new Voice = "Bass Voice" { b,4 }""",
...     name='Bass Staff'))
```

```
>>> f(score)
\new Score <<
    \context Staff = "Treble Staff" {
        \context Voice = "Treble Voice" {
            c'4
        }
    }
    \context Staff = "Bass Staff" {
        \context Voice = "Bass Voice" {
            b,4
        }
    }
>>
```

```
>>> for x in componenttools.Parentage(score): x
...
Score<<2>>
```

```
>>> for x in componenttools.Parentage(score['Bass Voice'][0]): x
...
Note('b,4')
Voice-"Bass Voice"{1}
Staff-"Bass Staff"{1}
Score<<2>>
```

Parentage is treated as a selection of the component's improper parentage.

Return parentage instance.

Read-only properties

Parentage.**component**
The component from which the selection was derived.

Parentage.**containment_signature**
New in version 1.1. Containment signature of component:

```
>>> score = Score(
... r"""\context Staff = "CustomStaff" { """
...     r"""\context Voice = "CustomVoice" { c' d' e' f' } }""")
>>> score.name = 'CustomScore'
```

```
>>> f(score)
\context Score = "CustomScore" <<
    \context Staff = "CustomStaff" {
        \context Voice = "CustomVoice" {
            c'4
            d'4
            e'4
            f'4
        }
    }
>>
```

```
>>> score.leaves[0].parentage.containment_signature
ContainmentSignature(Note-..., Voice-'CustomVoice', Staff-..., Score-'CustomScore')
```

Return containment signature object.

Parentage.**depth**
Length of proper parentage of component.

Return nonnegative integer.

Parentage.**is_orphan**
True when component has no parent. Otherwise false.

Return boolean.

Parentage.**music**
Read-only tuple of components in selection.

Parentage.**parent**
Parent of component or none when component is orphan.

Return component or none.

Parentage.**parentage_signature**
New in version 1.1. Parentage signature of component:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> staff = Staff([tuplet])
>>> note = staff.leaves[0]
```

```
>>> print note.parentage.parentage_signature
    staff: Staff-...
    self: Note-...
```

Return parentage signature.

Parentage.**root**
Last element in parentage.

Parentage.**score_index**
Score index of component:

```
>>> staff_1 = Staff(r"\times 2/3 { c'8 d'8 e'8 } \times 2/3 { f'8 g'8 a'8 }")
>>> staff_2 = Staff(r"\times 2/3 { b'8 c''8 d''8 }")
>>> score = Score([staff_1, staff_2])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \times 2/3 {
            c'8
            d'8
            e'8
        }
        \times 2/3 {
            f'8
            g'8
            a'8
        }
    }
    \new Staff {
        \times 2/3 {
            b'8
            c''8
            d''8
        }
    }
>>
```

```
>>> for leaf in score.leaves:
...     leaf, leaf.parentage.score_index
...
(Note("c'8"), (0, 0, 0))
(Note("d'8"), (0, 0, 1))
(Note("e'8"), (0, 0, 2))
(Note("f'8"), (0, 1, 0))
(Note("g'8"), (0, 1, 1))
(Note("a'8"), (0, 1, 2))
(Note("b'8"), (1, 0, 0))
(Note("c''8"), (1, 0, 1))
(Note("d''8"), (1, 0, 2))
```

Return tuple of zero or more nonnegative integers.

Parentage.**storage_format**
Storage format of Abjad object.

Return string.

Parentage.**timespan**
Read-only timespan of selection.

Parentage.**tuplet_depth**
Tuplet-depth of component:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> staff = Staff([tuplet])
>>> note = staff.leaves[0]
```

```
>>> note.parentage.tuplet_depth
1
```

```
>>> tuplet.parentage.tuplet_depth
0
```

```
>>> staff.parentage.tuplet_depth
0
```

Return nonnegative integer.

Special methods

`Parentage.`**`__add__`**(*expr*)

`Parentage.`**`__contains__`**(*expr*)

`Parentage.`**`__eq__`**(*expr*)

`Parentage.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Parentage.`**`__getitem__`**(*expr*)

`Parentage.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`Parentage.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Parentage.`**`__len__`**()

`Parentage.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Parentage.`**`__ne__`**(*expr*)

`Parentage.`**`__radd__`**(*expr*)

`Parentage.`**`__repr__`**()

3.3 Functions

3.3.1 componenttools.all_are_components

`componenttools.`**`all_are_components`**(*expr*, *klasses=None*)
New in version 1.1. True when elements in *expr* are all components:

```
>>> componenttools.all_are_components(3 * Note("c'4"))
True
```

Otherwise false:

```
>>> componenttools.all_are_components(['foo', 'bar'])
False
```

True when elements in *expr* are all *klasses*:

```
>>> componenttools.all_are_components(3 * Note("c'4"), klasses = Note)
True
```

Otherwise false:

```
>>> componenttools.all_are_components(['foo', 'bar'], klasses = Note)
False
```

Return boolean.

3.3.2 componenttools.all_are_components_in_same_thread

componenttools.**all_are_components_in_same_thread**(*expr*, *klasses=None*, *allow_orphans=True*)

New in version 1.1. True when elements in *expr* are all components in same thread. Otherwise false:

```
>>> voice = Voice("c'8 d'8 e'8")
>>> componenttools.all_are_components_in_same_thread(voice.leaves)
True
```

True when elements in *expr* are all *klasses* in same thread. Otherwise false:

```
>>> voice = Voice("c'8 d'8 e'8")
>>> componenttools.all_are_components_in_same_thread(voice.leaves, klasses=Note)
True
```

Return boolean.

3.3.3 componenttools.all_are_components_scalable_by_multiplier

componenttools.**all_are_components_scalable_by_multiplier**(*components*, *multiplier*)

New in version 1.1. True when *components* are all scalable by *multiplier*:

```
>>> components = [Note(0, (1, 8))]
>>> componenttools.all_are_components_scalable_by_multiplier(components, Multiplier(3, 2))
True
```

Otherwise false:

```
>>> components = [Note(0, (1, 8))]
>>> componenttools.all_are_components_scalable_by_multiplier(components, Multiplier(2, 3))
False
```

Return boolean.

3.3.4 componenttools.all_are_contiguous_components

componenttools.**all_are_contiguous_components**(*expr*, *klasses=None*, *allow_orphans=True*)

New in version 1.1. True when elements in *expr* are all contiguous components. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components(staff.leaves)
True
```

True when elements in *expr* are all contiguous *klasses*. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components(staff.leaves, klasses=Note)
True
```

Return boolean.

3.3.5 componenttools.all_are_contiguous_components_in_same_parent

componenttools.**all_are_contiguous_components_in_same_parent**(*expr*, *klasses=None*, *allow_orphans=True*)

New in version 1.1. True when elements in *expr* are all contiguous components in same parent. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components_in_same_parent(staff.leaves)
True
```

True when elements in *expr* are all contiguous *klasses* in same parent. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components_in_same_parent(staff.leaves, klasses=Note)
True
```

Return boolean.

3.3.6 componenttools.all_are_contiguous_components_in_same_thread

componenttools.**all_are_contiguous_components_in_same_thread**(*expr*, *klasses=None*, *allow_orphans=True*)

New in version 1.1. True when elements in *expr* are all contiguous components in same thread. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components_in_same_thread(staff.leaves)
True
```

True when elements in *expr* are all contiguous *klasses* in same thread. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8")
>>> componenttools.all_are_contiguous_components_in_same_thread(staff.leaves, klasses=Note)
True
```

Return boolean.

3.3.7 componenttools.all_are_thread_contiguous_components

componenttools.**all_are_thread_contiguous_components**(*expr*, *klasses=None*, *allow_orphans=True*)

New in version 1.1. True when elements in *expr* are all thread-contiguous components:

```
t = Voice(notetools.make_repeated_notes(4))
t.insert(2, Voice(notetools.make_repeated_notes(2)))
Container(t[:2])
Container(t[-2:])
pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)

\new Voice {
    {
        c'8
        d'8
    }
    \new Voice {
        e'8
        f'8
    }
```

```
        {
            g'8
            a'8
        }
    }

    assert _are_thread_contiguous_components(t[0:1] + t[-1:])
    assert _are_thread_contiguous_components(t[0][:] + t[-1:])
    assert _are_thread_contiguous_components(t[0:1] + t[-1][:])
    assert _are_thread_contiguous_components(t[0][:] + t[-1][:])
```

Return boolean.

Thread-contiguous components are, by definition, spannable.

3.3.8 componenttools.copy_and_partition_governed_component_subtree_by_leaf_counts

componenttools.**copy_and_partition_governed_component_subtree_by_leaf_counts**(*container*, *leaf_counts*)

New in version 1.1. Copy *container* and partition copy according to *leaf_counts*:

```
>>> voice = Voice(r"\times 2/3 { c'8 d'8 e'8 } \times 2/3 { f'8 g'8 a'8 }")
>>> beamtools.BeamSpanner(voice[0].leaves)
BeamSpanner(c'8, d'8, e'8)
>>> beamtools.BeamSpanner(voice[1].leaves)
BeamSpanner(f'8, g'8, a'8)
```

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
    }
    \times 2/3 {
        f'8 [
        g'8
        a'8 ]
    }
}
```

```
>>> result = componenttools.copy_and_partition_governed_component_subtree_by_leaf_counts(
... voice, [1, 2, 3])
```

```
>>> first, second, third = result
```

```
>>> f(first)
\new Voice {
    \times 2/3 {
        c'8 [ ]
    }
}
```

```
>>> f(second)
\new Voice {
    \times 2/3 {
        d'8 [
        e'8 ]
    }
}
```

```
>>> f(third)
\new Voice {
    \times 2/3 {
        f'8 [
        g'8
        a'8 ]
```

```
    }
}
```

Set *leaf_counts* to an iterable of zero or more positive integers.

Return a list of parts equal in length to that of *leaf_counts*.

3.3.9 componenttools.copy_components_and_covered_spanners

componenttools.**copy_components_and_covered_spanners**(*components*, *n=1*)

New in version 1.1. Copy *components* and covered spanners.

The *components* must be thread-contiguous.

Covered spanners are those spanners that cover *components*.

The steps taken in this function are as follows. Withdraw *components* from crossing spanners. Preserve spanners that *components* cover. Deep copy *components*. Reapply crossing spanners to source *components*. Return copied components with covered spanners.

```
>>> voice = Voice(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
>>> f(voice)
\new Voice {
    {
        \time 2/8
        c'8 [
        d'8
    }
    {
        e'8
        f'8 ]
    }
    {
        g'8
        a'8
    }
}
```

```
>>> result = componenttools.copy_components_and_covered_spanners(voice.leaves)
>>> result
(Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'8"), Note("a'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
    g'8
    a'8
}
```

```
>>> voice.leaves[0] is new_voice.leaves[0]
False
```

Copy *components* a total of *n* times.

```
>>> result = componenttools.copy_components_and_covered_spanners(voice.leaves[:2], n=3)
>>> result
(Note("c'8"), Note("d'8"), Note("c'8"), Note("d'8"), Note("c'8"), Note("d'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    c'8
    d'8
```

```
        c'8
        d'8
        c'8
        d'8
    }
```

Return new components.

3.3.10 componenttools.copy_components_and_fracture_crossing_spanners

componenttools.**copy_components_and_fracture_crossing_spanners**(*components*, *n=1*)

New in version 1.1. Copy *components* and fracture crossing spanners.

The *components* must be thread-contiguous.

The steps this function takes are as follows. Deep copy *components*. Deep copy spanners that attach to any component in *components*. Fracture spanners that attach to components not in *components*. Return Python list of copied components.

```
>>> voice = Voice(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
>>> f(voice)
\new Voice {
    {
        \time 2/8
        c'8 [
        d'8
    }
    {
        e'8
        f'8 ]
    }
    {
        g'8
        a'8
    }
}
```

```
>>> result = componenttools.copy_components_and_fracture_crossing_spanners(
...     voice.leaves[2:4])
>>> result
(Note("e'8"), Note("f'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    e'8 [
    f'8 ]
}
```

```
>>> voice.leaves[2] is new_voice.leaves[0]
False
```

Copy *components* a total of *n* times.

```
>>> result = componenttools.copy_components_and_fracture_crossing_spanners(
...     voice.leaves[2:4], n=3)
>>> result
(Note("e'8"), Note("f'8"), Note("e'8"), Note("f'8"), Note("e'8"), Note("f'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    e'8 [
    f'8 ]
    e'8 [
    f'8 ]
```

```
        e'8 [
        f'8 ]
    }
```

Return new components.

3.3.11 componenttools.copy_components_and_immediate_parent_of_first_component

componenttools.**copy_components_and_immediate_parent_of_first_component**(*components*)

New in version 1.1. Copy *components* and immediate parent of first component.

The *components* must be thread-contiguous.

Return in newly created container equal to type of first element in *copmonents*.

If the parent of the first element in *components* is a tuplet then insure that the tuplet multiplier of the function output equals the tuplet multiplier of the parent of the first element in *components*.

```
>>> voice = Voice(r"\times 2/3 { c'8 d' e' } \times 2/3 { f'8 g' a' }")
>>> voice.append(r"\times 2/3 { b'8 c'' d'' }")
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
```

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
    \times 2/3 {
        f'8 ]
        g'8
        a'8
    }
    \times 2/3 {
        b'8
        c''8
        d''8
    }
}
```

```
>>> leaves = voice.leaves[:2]
>>> componenttools.copy_components_and_immediate_parent_of_first_component(leaves)
Tuplet(2/3, [c'8, d'8])
```

Parent-contiguity is not required. Thread-contiguous *components* suffice.

```
>>> leaves = voice.leaves[:5]
>>> componenttools.copy_components_and_immediate_parent_of_first_component(leaves)
Tuplet(2/3, [c'8, d'8, e'8, f'8, g'8])
```

Note: this function copies only the *immediate parent* of the first element in *components*. This function ignores any further parentage of *components* above the immediate parent of *components*.

Todo

this function should (but does not) copy marks that attach to *components* and to the immediate parent of the first component; extend function to do so.

Return new components.

3.3.12 componenttools.copy_components_and_remove_spanners

componenttools.**copy_components_and_remove_spanners**(*components*, *n=1*)

New in version 1.1. Copy *components* and remove spanners.

The *components* must be thread-contiguous.

The steps taken by this function are as follows:

- Withdraw *components* from spanners.
- Deep copy unspanned *components*.
- Reapply spanners to *components*.
- Return unspanned copy.

Example:

```
>>> voice = Voice(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
>>> f(voice)
\new Voice {
    {
        \time 2/8
        c'8 [
        d'8
    }
    {
        e'8
        f'8 ]
    }
    {
        g'8
        a'8
    }
}
```

```
>>> result = componenttools.copy_components_and_remove_spanners(voice.leaves[2:4])
>>> result
(Note("e'8"), Note("f'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    e'8
    f'8
}
```

```
>>> voice.leaves[2] is new_voice.leaves[0]
False
```

Copy *components* a total of *n* times:

```
>>> result = componenttools.copy_components_and_remove_spanners(voice.leaves[2:4], n=3)
>>> result
(Note("e'8"), Note("f'8"), Note("e'8"), Note("f'8"), Note("e'8"), Note("f'8"))
```

```
>>> new_voice = Voice(result)
>>> f(new_voice)
\new Voice {
    e'8
    f'8
    e'8
    f'8
    e'8
    f'8
}
```

Return type equal to input type.

3.3.13 componenttools.copy_governed_component_subtree_by_leaf_range

componenttools.**copy_governed_component_subtree_by_leaf_range**(*component*, *start=0*, *stop=None*)

New in version 1.1. Copy governed *component* subtree by leaf range.

Governed subtree means *component* together with children of *component*.

Leaf range refers to the sequential parentage of *component* from *start* leaf index to *stop* leaf index:

```
>>> voice = Voice(r"\times 2/3 { c'8 d'8 e'8 } \times 2/3 { f'8 g'8 a'8 }")
>>> t = Staff([voice])
```

```
>>> f(t)
\new Staff {
    \new Voice {
        \times 2/3 {
            c'8
            d'8
            e'8
        }
        \times 2/3 {
            f'8
            g'8
            a'8
        }
    }
}
```

```
>>> u = componenttools.copy_governed_component_subtree_by_leaf_range(t, 1, 5)
>>> f(u)
\new Staff {
    \new Voice {
        \times 2/3 {
            d'8
            e'8
        }
        \times 2/3 {
            f'8
            g'8
        }
    }
}
```

Copy sequential containers in leaves' parentage up to the first parallel container in leaves' parentage.

Trim and shrink copied containers as necessary.

When *stop* is none copy all leaves from *start* forward.

Return new components.

3.3.14 componenttools.copy_governed_component_subtree_from_offset_to

componenttools.**copy_governed_component_subtree_from_offset_to**(*component*, *start=0*, *stop=None*)

New in version 1.1. Copy governed *component* subtree from *start* prolated duration to *stop* prolated duration.

Governed subtree refers to *component* together with the children of *component*:

```
>>> voice = Voice(r"c'8 d'8 \times 2/3 { e'8 f'8 g'8 }")
```

```
>>> f(voice)
\new Voice {
   c'8
   d'8
```

```
        \times 2/3 {
            e'8
            f'8
            g'8
        }
    }
```

```
>>> new = componenttools.copy_governed_component_subtree_from_offset_to(
...     voice, (0, 8), (3, 8))
```

```
>>> f(new)
\new Voice {
    c'8
    d'8
    \times 2/3 {
        e'8
        f'16
    }
}
```

Raise contiguity error if asked to slice a parallel container.

```
>>> staff = Staff(Voice("c'8 d'8") * 2)
>>> staff.is_parallel = True
>>> f(staff)
\new Staff <<
\new Voice {
    c'8
    d'8
}
\new Voice {
    c'8
    d'8
}
>>
```

Raise contiguity error when attempting to copy fleaves from parallel container.

But note that cases with `0 = start` work correctly:

```
>>> new = componenttools.copy_governed_component_subtree_from_offset_to(
...     voice, (0, 8), (1, 8))
```

```
>>> f(new)
\new Voice {
    c'8
}
```

Cases with `0 < start` do not work correctly:

```
>>> new = componenttools.copy_governed_component_subtree_from_offset_to(
...     voice, (1, 8), (2, 8))
```

```
>>> f(new)
\new Voice {
    c'8
    d'8
}
```

Create ad hoc tuplets as required:

```
>>> voice = Voice([Note("c'4")])
>>> new = componenttools.copy_governed_component_subtree_from_offset_to(
...     voice, 0, (1, 12))
```

```
>>> f(new)
\new Voice {
    \times 2/3 {
        c'8
    }
}
```

Function does NOT copy parentage of *component* when *component* is a leaf:

```
>>> voice = Voice([Note("c'4")])
>>> new_leaf = componenttools.copy_governed_component_subtree_from_offset_to(
...     voice[0], 0, (1, 8))
```

```
>>> f(new_leaf)
c'8
```

```
>>> new_leaf.parent is None
True
```

Return `(untrimmed_copy, first_dif, second_dif)`.

3.3.15 componenttools.extend_in_parent_of_component

`componenttools.`**`extend_in_parent_of_component`**(*component*, *new_components*, *left=False*, *grow_spanners=True*)

New in version 2.10. Extend *new_components* in parent of *component*.

Example 1. Extend *new_component* in parent of *component*. Grow spanners:

```
>>> voice = Voice("c'8 [ d'8 e'8 ]")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> new_components = [Note("c'8"), Note("d'8"), Note("e'8")]
>>> componenttools.extend_in_parent_of_component(
...     voice.leaves[-1], new_components, grow_spanners=True)
[Note("e'8"), Note("c'8"), Note("d'8"), Note("e'8")]
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    c'8
    d'8
    e'8 ]
}
```

Example 2. Extend *new_component* in parent of *component*. Do not grow spanners:

```
>>> voice = Voice("c'8 [ d'8 e'8 ]")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> new_components = [Note("c'8"), Note("d'8"), Note("e'8")]
>>> componenttools.extend_in_parent_of_component(
...     voice.leaves[-1], new_components, grow_spanners=False)
[Note("e'8"), Note("c'8"), Note("d'8"), Note("e'8")]
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8 ]
    c'8
```

```
        d'8
        e'8
}
```

Example 3. Extend *new_components* left in parent of *component*. Grow spanners:

```
>>> voice = Voice("c'8 [ d'8 e'8 ]")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8")]
>>> componenttools.extend_in_parent_of_component(
...     voice[0], notes, left=True, grow_spanners=True)
[Note("c'8"), Note("d'8"), Note("e'8"), Note("c'8")]
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    c'8
    d'8
    e'8 ]
}
```

Example 4. Extend *new_components* left in parent of *component*. Do not grow spanners:

```
>>> voice = Voice("c'8 [ d'8 e'8 ]")
```

```
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8")]
>>> componenttools.extend_in_parent_of_component(
...     voice[0], notes, left=True, grow_spanners=False)
[Note("c'8"), Note("d'8"), Note("e'8"), Note("c'8")]
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8
    e'8
    c'8 [
    d'8
    e'8 ]
}
```

Return *component* and *new_components* together in newly constructed list.

3.3.16 componenttools.get_component_in_expr_with_name

componenttools.**get_component_in_expr_with_name**(*expr*, *name*)
New in version 2.10. Get component in *expr* with *name*:

```
>>> voice_1 = Voice("c'4 d'4 e'4 f'4")
>>> voice_1.name = 'Top Voice'
>>> voice_2 = Voice(r'\clef "bass" c4 d4 e4 f4')
>>> voice_2.name = 'Bottom Voice'
>>> staff = Staff([voice_1, voice_2])
>>> f(staff)
\new Staff {
    \context Voice = "Top Voice" {
        c'4
        d'4
        e'4
        f'4
```

```
        }
        \context Voice = "Bottom Voice" {
            \clef "bass"
            c4
            d4
            e4
            f4
        }
    }
```

```
>>> voice = componenttools.get_component_in_expr_with_name(staff, 'Top Voice')
>>> f(voice)
\context Voice = "Top Voice" {
    c'4
    d'4
    e'4
    f'4
}
```

Return one component.

Raise missing component error when no named component is found.

Raise extra component error when more than one component with *name* is found.

3.3.17 componenttools.get_components_in_expr_with_name

componenttools.**get_components_in_expr_with_name**(*expr*, *name*)
New in version 2.9. Get components in *expr* with *name*:

```
>>> staff = Staff(r"\new Voice { c'8 d'8 } \new Voice { e'8 f'8 } \new Voice { g'4 }")
>>> staff[0].name = 'outer voice'
>>> staff[1].name = 'middle voice'
>>> staff[2].name = 'outer voice'

>>> f(staff)
\new Staff {
    \context Voice = "outer voice" {
        c'8
        d'8
    }
    \context Voice = "middle voice" {
        e'8
        f'8
    }
    \context Voice = "outer voice" {
        g'4
    }
}

>>> componenttools.get_components_in_expr_with_name(staff, 'outer voice')
[Voice-"outer voice"{2}, Voice-"outer voice"{1}]
```

```
>>> componenttools.get_components_in_expr_with_name(staff, 'middle voice')
[Voice-"middle voice"{2}]
```

Return list of zero or more components found.

3.3.18 componenttools.get_first_component_in_expr_with_name

componenttools.**get_first_component_in_expr_with_name**(*expr*, *name*)
New in version 1.1. Get first component in *expr* with *name*:

```
>>> flute_staff = Staff("c'8 d'8 e'8 f'8")
>>> flute_staff.name = 'Flute'
>>> violin_staff = Staff("c'8 d'8 e'8 f'8")
>>> violin_staff.name = 'Violin'
```

```
>>> staff_group = scoretools.StaffGroup([flute_staff, violin_staff])
>>> score = Score([staff_group])
```

```
>>> componenttools.get_first_component_in_expr_with_name(score, 'Violin')
Staff-"Violin"{4}
```

Return component.

3.3.19 componenttools.get_first_component_with_name_in_improper_parentage_of_compo

componenttools.**get_first_component_with_name_in_improper_parentage_of_component**(*component*, *name*)

New in version 2.0. Get first component with *name* in improper parentage of *component*:

```
>>> score = Score([Staff("c'4 d'4 e'4 f'4")])
>>> score.name = 'The Score'
```

```
>>> f(score)
\context Score = "The Score" <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
>>
```

```
>>> leaf = score.leaves[0]
```

```
>>> componenttools.get_first_component_with_name_in_improper_parentage_of_component(
...     leaf, 'The Score')
Score-"The Score"<<1>>
```

```
>>> componenttools.get_first_component_with_name_in_improper_parentage_of_component(
...     leaf, 'foo') is None
True
```

Return component or none.

3.3.20 componenttools.get_first_component_with_name_in_proper_parentage_of_compone

componenttools.**get_first_component_with_name_in_proper_parentage_of_component**(*component*, *name*)

New in version 2.0. Get first component with *name* in proper parentage of *component*:

```
>>> score = Score([Staff("c'4 d'4 e'4 f'4")])
>>> score.name = 'The Score'
```

```
>>> f(score)
\context Score = "The Score" <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
>>
```

```
>>> leaf = score.leaves[0]
```

```
>>> componenttools.get_first_component_with_name_in_proper_parentage_of_component(
...     leaf, 'The Score')
Score-"The Score"<<1>>
```

```
>>> componenttools.get_first_component_with_name_in_proper_parentage_of_component(
...     leaf, 'foo') is None
True
```

Return component or none.

3.3.21 componenttools.get_first_instance_of_klass_in_improper_parentage_of_component

componenttools.**get_first_instance_of_klass_in_improper_parentage_of_component**(*component*, *klass*)

New in version 2.0. Get first instance of *klass* in improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> componenttools.get_first_instance_of_klass_in_improper_parentage_of_component(
...     staff[0], Note)
Note("c'8")
```

Return component or none.

3.3.22 componenttools.get_first_instance_of_klass_in_proper_parentage_of_component

componenttools.**get_first_instance_of_klass_in_proper_parentage_of_component**(*component*, *klass*)

New in version 1.1. Get first instance of *klass* in proper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> componenttools.get_first_instance_of_klass_in_proper_parentage_of_component(
...     staff[0], Staff)
Staff{4}
```

Return component or none.

3.3.23 componenttools.get_improper_contents_of_component

componenttools.**get_improper_contents_of_component**(*component*)

New in version 2.9. Get improper contents of *component*:

```
>>> staff = Staff("c' d' e' f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> componenttools.get_improper_contents_of_component(staff)
[Staff{4}, Note("c'4"), Note("d'4"), Note("e'4"), Note("f'4")]
```

The functions works for both containers and leaves.

Return a list of *component* together with the proper contents of *component*.

3.3.24 componenttools.get_improper_descendents_of_component

componenttools.**get_improper_descendents_of_component**(*component*)

New in version 2.9. Get improper descendents of *component*:

```
>>> staff = Staff(r"c'4 \times 2/3 { d'8 e'8 f'8 }")
```

```
>>> f(staff)
\new Staff {
    c'4
    \times 2/3 {
        d'8
        e'8
        f'8
    }
}
```

```
>>> for x in componenttools.get_improper_descendents_of_component(staff):
...     x
...
Staff{2}
Note("c'4")
Tuplet(2/3, [d'8, e'8, f'8])
Note("d'8")
Note("e'8")
Note("f'8")
```

Function returns exactly the same components as `iterationtools.iterate_components_in_expr()`.

Return list of *component* together with proper descendents of *component*.

3.3.25 componenttools.get_improper_descendents_of_component_that_cross_offset

componenttools.**get_improper_descendents_of_component_that_cross_offset**(*component*, *prolated_offset*)

New in version 2.0. Get improper contents of *component* that cross *prolated_offset*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
}
```

Examples refer to the score above.

No components cross prolated offset `0`:

```
>>> componenttools.get_improper_descendents_of_component_that_cross_offset(staff, 0)
[]
```

Staff, measure and leaf cross prolated offset `1/16`:

```
>>> componenttools.get_improper_descendents_of_component_that_cross_offset(
...     staff, Duration(1, 16))
[Staff{2}, Measure(2/8, [c'8, d'8]), Note("c'8")]
```

Staff and measure cross prolated offset `1/8`:

```
>>> componenttools.get_improper_descendents_of_component_that_cross_offset(
...     staff, Duration(1, 8))
[Staff{2}, Measure(2/8, [c'8, d'8])]
```

Staff crosses prolated offset `1/4`:

```
>>> componenttools.get_improper_descendents_of_component_that_cross_offset(
...     staff, Duration(1, 4))
[Staff{2}]
```

No components cross prolated offset `99`:

```
>>> componenttools.get_improper_descendents_of_component_that_cross_offset(
...     staff, 99)
[]
```

Return list.

3.3.26 componenttools.get_improper_descendents_of_component_that_start_with_component

componenttools.**get_improper_descendents_of_component_that_start_with_component**(*component*)

New in version 2.9. Get improper contents of *component* that start with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d' } \new Voice { e' } >> f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'4
        }
        \new Voice {
            e'4
        }
    >>
    f'4
}
```

```
>>> componenttools.get_improper_descendents_of_component_that_start_with_component(
...     staff[1])
[<<Voice{1}, Voice{1}>>, Voice{1}, Note("d'4"), Voice{1}, Note("e'4")]
```

Return list of *component* together with improper contents that start with component.

3.3.27 componenttools.get_improper_descendents_of_component_that_stop_with_component

componenttools.**get_improper_descendents_of_component_that_stop_with_component**(*component*)

New in version 2.9. Get improper descendents of *component* that stop with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d' } \new Voice { e' } >> f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'4
        }
        \new Voice {
            e'4
        }
    >>
    f'4
}
```

```
>>> componenttools.get_improper_descendents_of_component_that_stop_with_component(staff)
[Staff{3}, Note("f'4")]
```

Return list of *component* together with proper contents that stop with *component*.

3.3.28 componenttools.get_improper_parentage_of_component

componenttools.**get_improper_parentage_of_component**(*component*)
New in version 1.1. Get improper parentage of *component*:

```
>>> tuplet = Tuplet(Fraction(2, 3), "c'8 d'8 e'8")
>>> staff = Staff([tuplet])
>>> note = staff.leaves[0]
```

```
>>> componenttools.get_improper_parentage_of_component(note)
(Note("c'8"), Tuplet(2/3, [c'8, d'8, e'8]), Staff{1})
```

Return tuple of zero or more components.

3.3.29 componenttools.get_improper_parentage_of_component_that_start_with_component

componenttools.**get_improper_parentage_of_component_that_start_with_component**(*component*)
New in version 2.9. Get improper parentage of *component* that start with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d' } \new Voice { e' } >> f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'4
        }
        \new Voice {
            e'4
        }
    >>
    f'4
}
```

```
>>> componenttools.get_improper_parentage_of_component_that_start_with_component(
...     staff.leaves[1])
[Note("d'4"), Voice{1}, <<Voice{1}, Voice{1}>>]
```

Return list of *component* together with proper parentage that start with *component*.

3.3.30 componenttools.get_improper_parentage_of_component_that_stop_with_component

componenttools.**get_improper_parentage_of_component_that_stop_with_component**(*component*)
New in version 2.9. Get improper parentage of *component* that stop with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d' } \new Voice { e' } >> f'")
```

```
f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'4
        }
        \new Voice {
            e'4
        }
    >>
    f'4
}
```

```
>>> componenttools.get_improper_parentage_of_component_that_stop_with_component(
...     staff.leaves[-1])
[Note("f'4"), Staff{3}]
```

Return list of *component* with proper parentage that stop with *component*.

3.3.31 componenttools.get_leftmost_components_with_total_duration_at_most

`componenttools.`**`get_leftmost_components_with_total_duration_at_most`**`(`*components*, *duration*`)`

New in version 2.0. Get leftmost components in *component* with prolated duration at most *duration*.

Return tuple of `components[:i]` together with the prolated duration of `components[:i]`:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> componenttools.get_leftmost_components_with_total_duration_at_most(
...     voice[:], Duration(1, 4))
([Note("c'8"), Note("d'8")], Duration(1, 4))
```

Maximize `i` such that the prolated duration of `components[:i]` is no greater than *duration*.

Input *components* must be thread-contiguous.

Todo

implement `componenttools.list_leftmost_components_with_prolated_duration_at_least()`.

Todo

implement `componenttools.list_rightmost_components_with_prolated_duration_at_most()`.

Todo

implement `componenttools.list_rightmost_components_with_prolated_duration_at_least()`.

Return components.

3.3.32 componenttools.get_lineage_of_component

`componenttools.`**`get_lineage_of_component`**`(`*component*`)`

New in version 2.9. Get lineage of *component*:

```
>>> staff = Staff(r"c'4 \times 2/3 { d'8 e'8 f'8 }")
```

```
f(staff)
\new Staff {
    c'4
    \times 2/3 {
        d'8
        e'8
        f'8
    }
}
```

```
componenttools.get_lineage_of_component(staff[1])
[Staff{2}, Tuplet(2/3, [d'8, e'8, f'8]), Note("d'8"), Note("e'8"), Note("f'8")]
```

Return list of parentage, component and descendents.

3.3.33 componenttools.get_lineage_of_component_that_start_with_component

componenttools.**get_lineage_of_component_that_start_with_component**(*component*)

New in version 2.9. Get lineage of *component* that start with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d'8 e'8 } \new Voice { d''8 e''8 } >> f'4")
```

```
>>> f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'8
            e'8
        }
        \new Voice {
            d''8
            e''8
        }
    >>
    f'4
}
```

```
>>> staff[1][0]
Voice{2}
```

```
>>> componenttools.get_lineage_of_component_that_start_with_component(staff[1][0])
[<<Voice{2}, Voice{2}>>, Voice{2}, Note("d'8")]
```

Return list of all components in the lineage of *component* that start with *component*.

The list always includes *component*.

3.3.34 componenttools.get_lineage_of_component_that_stop_with_component

componenttools.**get_lineage_of_component_that_stop_with_component**(*component*)

New in version 2.9. Get lineage of *component* that stop with *component*:

```
>>> staff = Staff(r"c' << \new Voice { d'8 e'8 } \new Voice { d''8 e''8 } >> f'4")
```

```
>>> f(staff)
\new Staff {
    c'4
    <<
        \new Voice {
            d'8
            e'8
        }
        \new Voice {
            d''8
            e''8
        }
    >>
    f'4
}
```

```
>>> staff[1][0]
Voice{2}
```

```
>>> componenttools.get_lineage_of_component_that_stop_with_component(staff[1][0])
[<<Voice{2}, Voice{2}>>, Voice{2}, Note("e'8")]
```

Return list of all components in the lineage of *component* that stop with *component*.

The list always includes *component*.

3.3.35 componenttools.get_most_distant_sequential_container_in_improper_parentage_of_

componenttools.**get_most_distant_sequential_container_in_improper_parentage_of_component**
New in version 2.9. Get first sequential container in the improper parentage of *component* such that the parent of sequential container is either a parellel container or else none:

```
>>> voice_1 = Voice("c'8 d'8")
>>> voice_2 = Voice("e'8 f'8")
>>> t = Voice([Container([voice_1, voice_2])])
>>> t[0].is_parallel = True
>>> t[0][0].name = 'voice 1'
>>> t[0][1].name = 'voice 2'
```

```
>>> f(t)
\new Voice {
    <<
        \context Voice = "voice 1" {
            c'8
            d'8
        }
        \context Voice = "voice 2" {
            e'8
            f'8
        }
    >>
}
```

```
>>> note = t.leaves[1]
```

```
>>> componenttools.get_most_distant_sequential_container_in_improper_parentage_of_component(
...     note)
Voice-"voice 1"{2}
```

Return none when no such container exists in the improper parentage of *component*.

3.3.36 componenttools.get_nth_component_in_expr

componenttools.**get_nth_component_in_expr**(*expr*, *klasses*, *n=0*)
New in version 1.1. Get component *n* in the *klasses* of *expr*:

```
>>> staff = Staff([])
>>> durations = [Duration(n, 16) for n in range(1, 5)]
>>> notes = notetools.make_notes([0, 2, 4, 5], durations)
>>> rests = resttools.make_rests(durations)
>>> leaves = sequencetools.interlace_sequences(notes, rests)
>>> staff.extend(leaves)
```

```
>>> f(staff)
\new Staff {
    c'16
    r16
    d'8
    r8
    e'8.
    r8.
    f'4
    r4
}
```

```
>>> for n in range(4):
...         componenttools.get_nth_component_in_expr(staff, Note, n)
...
Note("c'16")
Note("d'8")
Note("e'8.")
Note("f'4")
```

```
>>> for n in range(4):
...         componenttools.get_nth_component_in_expr(staff, Rest, n)
...
Rest('r16')
Rest('r8')
Rest('r8.')
Rest('r4')
```

```
>>> componenttools.get_nth_component_in_expr(staff, Staff)
Staff{8}
```

Read right-to-left for negative values of *n*:

```
>>> for n in range(3, -1, -1):
...         componenttools.get_nth_component_in_expr(staff, Rest, n)
...
Rest('r4')
Rest('r8.')
Rest('r8')
Rest('r16')
```

Return component or none.

3.3.37 componenttools.get_nth_component_in_time_order_from_component

componenttools.**get_nth_component_in_time_order_from_component**(*component*, *n*)

New in version 2.9. Get nth component from *component* in temporal order:

```
>>> staff = Staff(r"c'4 \times 2/3 { d'8 e'8 f'8 } g'2")
```

```
>>> f(staff)
\new Staff {
    c'4
    \times 2/3 {
        d'8
        e'8
        f'8
    }
    g'2
}
```

```
>>> staff.leaves[1]
Note("d'8")
```

Return component right of *component* for positive *n*:

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], 1)
Note("e'8")
```

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], 2)
Note("f'8")
```

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], 3)
Note("g'2")
```

Return component left of *component* for negative *n*:

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], -1)
Note("c'4")
```

Return *component* when *n* is `0`:

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], 0)
Note("d'8")
```

Return none when *n* is out of range:

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff.leaves[1], 99) is None
True
```

Return none when *component* has no parent:

```
>>> componenttools.get_nth_component_in_time_order_from_component(
...     staff, 1) is None
True
```

Return component or none.

3.3.38 componenttools.get_nth_namesake_from_component

componenttools.**get_nth_namesake_from_component**(*component*, *n*)
New in version 2.0. For positive *n*, return namesake to the right of *component*:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> componenttools.get_nth_namesake_from_component(t[1], 1)
Note("e'8")
```

For negative *n*, return namesake to the left of *component*:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> componenttools.get_nth_namesake_from_component(t[1], -1)
Note("c'8")
```

Return *component* when *n* is zero:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> componenttools.get_nth_namesake_from_component(t[1], 0)
Note("d'8")
```

Return component or none.

3.3.39 componenttools.get_nth_sibling_from_component

componenttools.**get_nth_sibling_from_component**(*component*, *n*)
New in version 2.9. Get nth sibling from *component*:

```
>>> staff = Staff("c' d' e' f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> staff[1]
Note("d'4")
```

Return sibling to the right of *component* for positive *n*:

```
>>> componenttools.get_nth_sibling_from_component(staff[1], 1)
Note("e'4")
```

Return sibling to the left of *component* for negative *n*:

```
>>> componenttools.get_nth_sibling_from_component(staff[1], -1)
Note("c'4")
```

Return *component* when *n* is `0`:

```
>>> componenttools.get_nth_sibling_from_component(staff[1], 0)
Note("d'4")
```

Return none when *n* is out of range:

```
>>> componenttools.get_nth_sibling_from_component(staff[1], 99) is None
True
```

Return none when *component* has no parent:

```
>>> componenttools.get_nth_sibling_from_component(staff, 1) is None
True
```

Return component or none.

3.3.40 componenttools.get_parent_and_start_stop_indices_of_components

componenttools.**get_parent_and_start_stop_indices_of_components**(*components*)

New in version 1.1. Get parent and start / stop indices of *components*:

```
>>> t = Staff("c'8 d'8 e'8 f'8 g'8 a'8")
```

```
>>> f(t)
\new Staff {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
}
```

```
>>> leaves = t[-2:]
>>> leaves
Selection(Note("g'8"), Note("a'8"))
>>> componenttools.get_parent_and_start_stop_indices_of_components(leaves)
(Staff{6}, 4, 5)
```

Return parent / start index / stop index triple. Return parent as component or none. Return nonnegative integer start index and nonnegative index stop index.

3.3.41 componenttools.get_proper_contents_of_component

componenttools.**get_proper_contents_of_component**(*component*)

New in version 2.9. Get proper contents of *component*:

```
>>> staff = Staff("c' d' e' f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> componenttools.get_proper_contents_of_component(staff)
[Note("c'4"), Note("d'4"), Note("e'4"), Note("f'4")]
```

The function works on leaves:

```
>>> componenttools.get_proper_contents_of_component(staff[0])
[]
```

Return list of the proper contents of component.

3.3.42 componenttools.get_proper_descendents_of_component

componenttools.**get_proper_descendents_of_component**(*component*)

New in version 2.9. Get proper descendents of *component*:

```
>>> staff = Staff(r"c'4 \times 2/3 { d'8 e'8 f'8 }")
```

```
>>> f(staff)
\new Staff {
    c'4
    \times 2/3 {
        d'8
        e'8
        f'8
    }
}
```

```
>>> componenttools.get_proper_descendents_of_component(staff)
[Note("c'4"), Tuplet(2/3, [d'8, e'8, f'8]), Note("d'8"), Note("e'8"), Note("f'8")]
```

Return list of proper descendents of *component*.

3.3.43 componenttools.get_proper_parentage_of_component

componenttools.**get_proper_parentage_of_component**(*component*)

New in version 1.1. Get proper parentage of *component*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> staff = Staff([tuplet])
>>> note = staff.leaves[0]
>>> componenttools.get_proper_parentage_of_component(note)
(FixedDurationTuplet(1/4, [c'8, d'8, e'8]), Staff{1})
```

Return tuple of zero or more components.

3.3.44 componenttools.is_immediate_temporal_successor_of_component

componenttools.**is_immediate_temporal_successor_of_component**(*component*, *expr*)

New in version 2.9. True when *expr* is immediate temporal successor of *component*.

Otherwise false.

3.3.45 componenttools.move_component_subtree_to_right_in_immediate_parent_of_compo

componenttools.**move_component_subtree_to_right_in_immediate_parent_of_component**(*component*

New in version 2.0. Move *component* subtree to right in immediate parent of *component*:

```
>>> voice = Voice("c'8 [ d'8 ] e'8 [ f'8 ]")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> componenttools.move_component_subtree_to_right_in_immediate_parent_of_component(
... voice[1])
```

```
>>> f(voice)
\new Voice {
    c'8 [
    e'8 ]
    d'8 [
    f'8 ]
}
```

Return none.

Todo

add `n = 1` keyword to generalize flipped distance.

Todo

make `componenttools.move_component_subtree_to_right_in_immediate_parent_of_component` work when spanners attach to children of component:

```
>>> voice = Voice(r"\times 2/3 { c'8 [ d'8 e'8 } \times 2/3 { f'8 ] g'8 a'8 }")
```

```
>>> componenttools.move_component_subtree_to_right_in_immediate_parent_of_component(
... voice[0])
```

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        f'8 ]
        g'8
        a'8
    }
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
}
```

```
>>> wellformednesstools.is_well_formed_component(voice)
False
```

Preserve spanners.

3.3.46 componenttools.move_parentage_and_spanners_from_components_to_components

`componenttools.`**`move_parentage_and_spanners_from_components_to_components`**(*donors*, *recipients*)

New in version 1.1. Move parentage and spanners from *donors* to *recipients*.

Give everything from donors to recipients. Almost exactly the same as container setitem logic. This helper works with orphan donors. Container setitem logic can not work with orphan donors.

Return donors.

3.3.47 componenttools.partition_components_by_durations_exactly

componenttools.**partition_components_by_durations_exactly**(*components*, *durations*, *cyclic=False*, *in_seconds=False*, *overhang=False*)

New in version 1.1.

3.3.48 componenttools.partition_components_by_durations_not_greater_than

componenttools.**partition_components_by_durations_not_greater_than**(*components*, *durations*, *cyclic=False*, *in_seconds=False*, *overhang=False*)

New in version 1.1.

3.3.49 componenttools.partition_components_by_durations_not_less_than

componenttools.**partition_components_by_durations_not_less_than**(*components*, *durations*, *cyclic=False*, *in_seconds=False*, *overhang=False*)

New in version 1.1. Partition *components* by *durations*.

Example 1. Partition *components* cyclically by prolated *durations*. Keep overhang:

```
>>> string = "abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 || 2/8 b'8 c''8 |"
>>> staff = Staff(string)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
    }
}
```

```
>>> parts = componenttools.partition_components_by_durations_not_less_than(
...     staff.leaves, [Duration(3, 16), Duration(1, 16)], cyclic=True, overhang=True)
```

```
>>> for part in parts:
...     part
...
[Note("c'8"), Note("d'8")]
[Note("e'8")]
[Note("f'8"), Note("g'8")]
```

```
[Note("a'8")]
[Note("b'8"), Note("c''8")]
```

Return list of lists.

Function works not just on components but on any durated objects including spanners.

3.3.50 componenttools.remove_component_subtree_from_score_and_spanners

componenttools.**remove_component_subtree_from_score_and_spanners**(*components*)

New in version 1.1. Example 1. Remove one leaf from score:

```
>>> voice = Voice("c'8 [ { d'8 e'8 } f'8 ]")
>>> spannertools.GlissandoSpanner(voice.leaves)
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    {
        d'8 \glissando
        e'8 \glissando
    }
    f'8 ]
}
```

```
>>> componenttools.remove_component_subtree_from_score_and_spanners(voice.leaves[1:2])
(Note("d'8"),)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    {
        e'8 \glissando
    }
    f'8 ]
}
```

Example 2. Remove contiguous leaves from score:

```
>>> voice = Voice("c'8 [ { d'8 e'8 } f'8 ]")
>>> spannertools.GlissandoSpanner(voice.leaves)
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    {
        d'8 \glissando
        e'8 \glissando
    }
    f'8 ]
}
```

```
>>> componenttools.remove_component_subtree_from_score_and_spanners(voice.leaves[:2])
(Note("c'8"), Note("d'8"))
```

```
>>> f(voice)
\new Voice {
    {
        e'8 [ \glissando
    }
    f'8 ]
}
```

Example 3. Remove noncontiguous leaves from score:

```
>>> voice = Voice("c'8 [ { d'8 e'8 } f'8 ]")
>>> spannertools.GlissandoSpanner(voice.leaves)
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    {
        d'8 \glissando
        e'8 \glissando
    }
    f'8 ]
}
```

```
>>> componenttools.remove_component_subtree_from_score_and_spanners(
... [voice.leaves[0], voice.leaves[2]])
[Note("c'8"), Note("e'8")]
```

```
>>> f(voice)
\new Voice {
    { d'8 [ \glissando
    }
    f'8 ]
}
```

Example 4. Remove container from score:

```
>>> voice = Voice("c'8 [ { d'8 e'8 } f'8 ]")
>>> spannertools.GlissandoSpanner(voice.leaves)
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    {
        d'8 \glissando
        e'8 \glissando
    }
    f'8 ]
}
```

```
>>> componenttools.remove_component_subtree_from_score_and_spanners(voice[1:2])
Selection({d'8, e'8},)
```

```
>>> f(voice)
\new Voice {
    c'8 [ \glissando
    f'8 ]
}
```

Withdraw *components* and children of *components* from spanners.

Return either tuple or list of *components* and children of *components*.

Todo

regularize return value of function.

3.3.51 componenttools.replace_components_with_children_of_components

componenttools.**replace_components_with_children_of_components**(*components*)

New in version 1.1. Remove arbitrary *components* from score but retain children of *components* in score:

```
>>> staff = Staff(Container(notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> spannertools.SlurSpanner(staff[:])
```

```
SlurSpanner({c'8, d'8}, {e'8, f'8})
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    {
        c'8 [ (
        d'8
    }
    {
        e'8
        f'8 ] )
    }
}
```

```
>>> componenttools.replace_components_with_children_of_components(staff[0:1])
Selection({},)
```

```
>>> f(staff)
\new Staff {
    c'8 [ (
    d'8
    {
        e'8
        f'8 ] )
    }
}
```

Return *components*.

3.3.52 componenttools.shorten_component_by_duration

componenttools.**shorten_component_by_duration**(*component*, *duration*)

New in version 2.0. Shorten *component* by prolated *duration*.

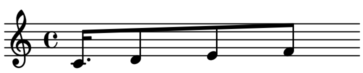
Example 1. Shorten container and produce note with dotted duration:

```
>>> staff = Staff("c'8 [ d'8 e'8 f'8 ]")
```

```
>>> componenttools.shorten_component_by_duration(staff, Duration(1, 32))
```

```
>>> f(staff)
\new Staff {
    c'16. [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(staff)
```



Example 2. Shorten container and produce note with tied duration:

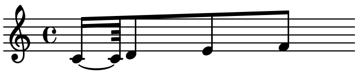
```
>>> staff = Staff("c'8 [ d'8 e'8 f'8 ]")
```

```
>>> componenttools.shorten_component_by_duration(staff, Duration(3, 64))
```

```
>>> f(staff)
\new Staff {
    c'16 [ ~
    c'64
    d'8
    e'8
```

```
        f'8 ]
    }
```

```
>>> show(staff)
```



Example 3. Shorten container and produce note with non-power-of-two duration:

```
>>> staff = Staff("c'8 [ d'8 e'8 f'8 ]")
```

```
>>> componenttools.shorten_component_by_duration(staff, Duration(1, 24))
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
    }
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(staff)
```

Return none.

3.3.53 componenttools.split_component_at_offset

componenttools.**split_component_at_offset**(*component*, *offset*, *fracture_spanners=False*, *tie_split_notes=True*, *tie_split_rests=False*)

New in version 1.1. Split *component* at *offset*.

Example 1. Split *component* at *offset*. Don't fracture spanners:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

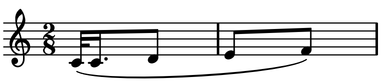
```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```

```
>>> halves = componenttools.split_component_at_offset(
... staff.leaves[0], (1, 32), fracture_spanners=False, tie_split_notes=False)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'32 [ (
        c'16.
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```



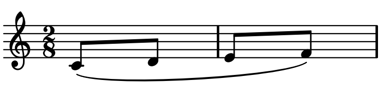
Example 2. Split component at offset at fracture crossing spanners:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

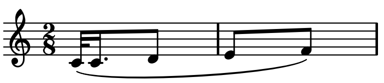
```
>>> show(staff)
```



```
>>> halves = componenttools.split_component_at_offset(
... staff.leaves[0], (1, 32), fracture_spanners=True, tie_split_notes=False)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'32 [ ( )
        c'16. (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```



Return pair of left and right part-lists.

3.3.54 componenttools.split_components_at_offsets

componenttools.**split_components_at_offsets**(*components*, *offsets*, *fracture_spanners=False*, *cyclic=False*, *tie_split_notes=True*, *tie_split_rests=False*)

New in version 2.0. Example 1. Split components cyclically and do not fracture crossing spanners:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

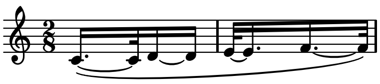
```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```

```
>>> componenttools.split_components_at_offsets(
...     staff.leaves, [Duration(3, 32)], cyclic=True)
[[Note("c'16.")], [Note("c'32"), Note("d'16")],
[Note("d'16"), Note("e'32")], [Note("e'16.")], [Note("f'16.")], [Note("f'32")]]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'16. [ ( ~
        c'32
        d'16 ~
        d'16 ]
    }
    {
        e'32 [ ~
        e'16.
        f'16. ~
        f'32 ] )
    }
}
```

```
>>> show(staff)
```



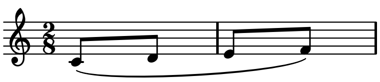
Example 2. Split components cyclically and fracture spanners:

```
>>> staff = Staff("abj: | 2/8 c’8 d’8 || 2/8 e’8 f’8 |")
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c’8, d’8, e’8, f’8)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c’8 [ (
        d’8 ]
    }
    {
        e’8 [
        f’8 ] )
    }
}
```

```
>>> show(staff)
```



```
>>> result = componenttools.split_components_at_offsets(
... staff.leaves, [Duration(3, 32)], cyclic=True, fracture_spanners=True)
```

```
>>> result
[[Note("c’16.")], [Note("c’32"), Note("d’16")], [Note("d’16"), Note("e’32")],
[Note("e’16.")], [Note("f’16.")], [Note("f’32")]]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c’16. [ ( ) ~
        c’32 (
        d’16 ) ~
        d’16 ] (
    }
    {
        e’32 [ ) ~
        e’16. (
        f’16. ) ~
        f’32 ] ( )
    }
}
```

```
>>> show(staff)
```

Example 3. Split components once and do not fracture crossing spanners:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

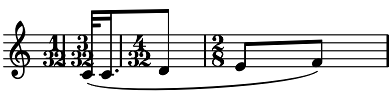
```
>>> show(staff)
```

```
>>> offsets = [Duration(1, 32), Duration(3, 32), Duration(5, 32)]
```

```
>>> shards = componenttools.split_components_at_offsets(
... staff[:1], offsets, cyclic=False, fracture_spanners=False, tie_split_notes=False)
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/32
        c'32 [ (
    }
    {
        \time 3/32
        c'16.
    }
    {
        \time 4/32
        d'8 ]
    }
    {
        \time 2/8
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```



Example 4. Split components once and fracture crossing spanners:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner(|2/8(2)|)
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

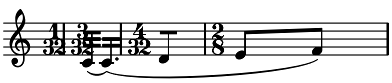
```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```

```
>>> offsets = [Duration(1, 32), Duration(3, 32), Duration(5, 32)]
>>> shards = componenttools.split_components_at_offsets(
... staff[:1], offsets, cyclic=False, fracture_spanners=True, tie_split_notes=False)
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/32
        c'32 [ ] ( )
    }
    {
        \time 3/32
        c'16. [ ] ( )
    }
    {
        \time 4/32
        d'8 [ ] (
    }
    {
        \time 2/8
        e'8 [
        f'8 ] )
    }
}
```

```
>>> show(staff)
```



Example 5. Split tupletted components once and fracture crossing spanners:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 e'8 } \times 2/3 { f'8 g'8 a'8 }")
```

```
>>> beamtools.BeamSpanner(staff[0])
BeamSpanner({c'8, d'8, e'8})
>>> beamtools.BeamSpanner(staff[1])
BeamSpanner({f'8, g'8, a'8})
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8, g'8, a'8)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [ (
        d'8
        e'8 ]
    }
    \times 2/3 {
        f'8 [
        g'8
```

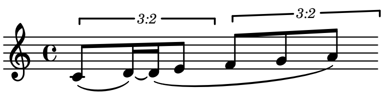
```
        a'8 ] )
    }
}
```

```
>>> show(staff)
```


```
>>> offsets = [(1, 8)]
>>> shards = componenttools.split_components_at_offsets(
... staff.leaves, offsets, cyclic=False, fracture_spanners=True, tie_split_notes=True)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [ (
        d'16 ) ~
        d'16 (
        e'8 ]
    }
    \times 2/3 {
        f'8 [
        g'8
        a'8 ] )
    }
}
```

```
>>> show(staff)
```



Return list of newly split shards.

Note: Add tests of tupletted notes and rests.

Note: Add examples that show mark and context mark handling.

Note: Add example showing grace and after grace handling.

3.3.55 componenttools.sum_duration_of_components

componenttools.**sum_duration_of_components**(*components*, *preprolated=False*, *in_seconds=False*)

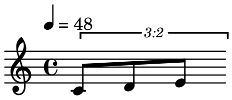
New in version 1.1. Sum duration of *components*:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> staff = Staff([tuplet])
>>> score = Score([staff])
>>> contexttools.TempoMark(Duration(1, 4), 48)(tuplet[0])
TempoMark(Duration(1, 4), 48)(c'8)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \times 2/3 {
            \tempo 4=48
            c'8
```

```
                d'8
                e'8
            }
        }
>>
```

```
>>> show(score)
```



Example 1. Sum duration of components:

```
>>> componenttools.sum_duration_of_components(tuplet[:])
Duration(1, 4)
```

Example 2. Sum preprolated duration of components:

```
>>> componenttools.sum_duration_of_components(tuplet[:], preprolated=True)
Duration(3, 8)
```

Example 3. Sum duration of components in seconds:

```
>>> componenttools.sum_duration_of_components(tuplet[:], in_seconds=True)
Duration(5, 4)
```

Return duration.

3.3.56 componenttools.yield_components_grouped_by_duration

componenttools.**yield_components_grouped_by_duration**(*components*)

New in version 2.0. Example 1. Yield topmost components grouped by prolated duration:

```
>>> staff = Staff(r"\times 2/3 { c'4 c'4 c'8 c'16 c'16 } c'16 c'16 c'8 c'8")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'4
        c'4
        c'8
        c'16
        c'16
    }
    c'16
    c'16
    c'8
    c'8
}
```

```
>>> show(staff)
```


```
>>> for x in componenttools.yield_components_grouped_by_duration(staff):
...     x
...
(Tuplet(2/3, [c'4, c'4, c'8, c'16, c'16]),)
(Note("c'16"), Note("c'16"))
(Note("c'8"), Note("c'8"))
```

Example 2. Yield topmost components grouped by prolated duration.

Note that function treats input as a flat sequence and attempts no navigation of the score tree. But it's possible to group components lower in the score tree by passing the output of a component iterator as input to this function:

```
>>> staff = Staff(r"\times 2/3 { c'4 c'4 c'8 c'16 c'16 } c'16 c'16 c'8 c'8")
```

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
>>> for x in componenttools.yield_components_grouped_by_duration(leaves):
...     x
(Note("c'4"), Note("c'4"))
(Note("c'8"),)
(Note("c'16"), Note("c'16"))
(Note("c'16"), Note("c'16"))
(Note("c'8"), Note("c'8"))
```

Return generator.

Note: Might be best to add `in_sequence` or `topmost` to the name of this function.

3.3.57 componenttools.yield_components_grouped_by_preprolated_duration

`componenttools.`**`yield_components_grouped_by_preprolated_duration`**(*components*)

New in version 2.0. Example 1. Yield topmost components grouped by preprolated duration:

```
>>> staff = Staff(r"\times 2/3 { c'4 c'4 c'8 c'16 c'16 } c'16 c'16 c'8 c'8")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'4
        c'4
        c'8
        c'16
        c'16
    }
    c'16
    c'16
    c'8
    c'8
}
```

```
>>> for x in componenttools.yield_components_grouped_by_preprolated_duration(staff):
...     x
...
(Tuplet(2/3, [c'4, c'4, c'8, c'16, c'16]),)
(Note("c'16"), Note("c'16"))
(Note("c'8"), Note("c'8"))
```

Example 2. Yield topmost components grouped by preprolated duration.

Note that function treats input as a flat sequence and attempts no navigation of the score tree. But it's possible to group components lower in the score tree by passing the output of a component iterator as input to this function:

```
>>> staff = Staff(r"\times 2/3 { c'4 c'4 c'8 c'16 c'16 } c'16 c'16 c'8 c'8")
```

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
>>> for x in componenttools.yield_components_grouped_by_preprolated_duration(leaves):
...     x
(Note("c'4"), Note("c'4"))
(Note("c'8"),)
(Note("c'16"), Note("c'16"), Note("c'16"), Note("c'16"))
(Note("c'8"), Note("c'8"))
```

Return generator.

Note: Might be best to add `in_sequence` or `topmost` to the name of this function.

3.3.58 componenttools.yield_groups_of_mixed_klasses_in_sequence

`componenttools.yield_groups_of_mixed_klasses_in_sequence`(*sequence*, *klasses*)
New in version 2.0. Example 1. Yield groups of notes and chords at only the top level of score:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 r8 } \times 2/3 { r8 <e' g'>8 <f' a'>8 }")
>>> staff.extend("g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        r8
    }
    \times 2/3 {
        r8
        <e' g'>8
        <f' a'>8
    }
    g'8
    a'8
    r8
    r8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(
...     staff, (Note, Chord)):
...     group
(Note("g'8"), Note("a'8"))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Example 2. Yield groups of notes and chords at all levels of score:

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
```

```
>>> for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(
...     leaves, (Note, Chord)):
...     group
(Note("c'8"), Note("d'8"))
(Chord("<e' g'>8"), Chord("<f' a'>8"), Note("g'8"), Note("a'8"))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Return generator.

3.3.59 componenttools.yield_topmost_components_grouped_by_type

`componenttools.yield_topmost_components_grouped_by_type`(*expr*)
New in version 2.0. Example 1. Yield topmost components in *expr* grouped by type:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 r8 } \times 2/3 { r8 <e' g'>8 <f' a'>8 }")
>>> staff.extend("g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        r8
    }
    \times 2/3 {
        r8
```

```
            <e' g'>8
            <f' a'>8
        }
        g'8
        a'8
        r8
        r8
        <b' d''>8
        <c'' e''>8
    }
```

```
>>> for x in componenttools.yield_topmost_components_grouped_by_type(staff):
...     x
(Tuplet(2/3, [c'8, d'8, r8]), Tuplet(2/3, [r8, <e' g'>8, <f' a'>8]))
(Note("g'8"), Note("a'8"))
(Rest('r8'), Rest('r8'))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Example 2. Yield leaves at all score levels in *expr* grouped by type:

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
```

```
>>> for x in componenttools.yield_topmost_components_grouped_by_type(leaves):
...     x
(Note("c'8"), Note("d'8"))
(Rest('r8'), Rest('r8'))
(Chord("<e' g'>8"), Chord("<f' a'>8"))
(Note("g'8"), Note("a'8"))
(Rest('r8'), Rest('r8'))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Return generator.

3.3.60 componenttools.yield_topmost_components_of_klass_grouped_by_type

componenttools.**yield_topmost_components_of_klass_grouped_by_type**(*expr*, *klass*)

New in version 2.0. Example 1. Yield runs of topmost notes in *expr*:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 r8 } \times 2/3 { r8 <e' g'>8 <f' a'>8 }")
>>> staff.extend("g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        r8
    }
    \times 2/3 {
        r8
        <e' g'>8
        <f' a'>8
    }
    g'8
    a'8
    r8
    r8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for group in componenttools.yield_topmost_components_of_klass_grouped_by_type(
...     staff, Note):
...     group
(Note("g'8"), Note("a'8"))
```

Example 2. Yield runs of notes at all levels in *expr*:

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
```

```
>>> for group in componenttools.yield_topmost_components_of_klass_grouped_by_type(
...     leaves, Note):
...     group
(Note("c'8"), Note("d'8"))
(Note("g'8"), Note("a'8"))
```

Return generator.

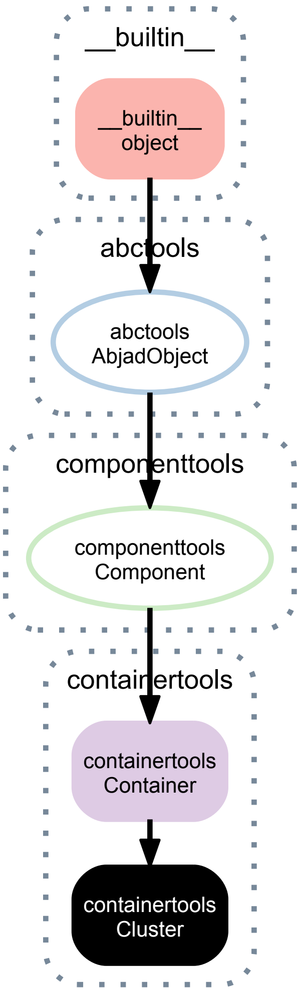
CHAPTER

FOUR

CONTAINERTOOLS

4.1 Concrete Classes

4.1.1 containertools.Cluster



class `containertools.Cluster` (*music=None*, ***kwargs*)

New in version 1.1. Abjad model of a tone cluster container:

```
>>> cluster = containertools.Cluster("c'8 <d' g'>8 b'8")
```

```
>>> cluster
Cluster(c'8, <d' g'>8, b'8)
```

```
>>> f(cluster)
\makeClusters {
    c'8
    <d' g'>8
    b'8
}
```

```
>>> show(cluster)
```

Return cluster object.

Read-only properties

`Cluster.`**contents_duration**

`Cluster.`**descendants**
Read-only reference to component descendants score selection.

`Cluster.`**duration**

`Cluster.`**duration_in_seconds**

`Cluster.`**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`Cluster.`**lilypond_format**

`Cluster.`**lineage**
Read-only reference to component lineage score selection.

`Cluster.`**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`Cluster.`**override**
Read-only reference to LilyPond grob override component plug-in.

`Cluster.`**parent**

`Cluster.`**parentage**
Read-only reference to component parentage score selection.

`Cluster.`**preprolated_duration**

`Cluster.`**prolation**

`Cluster.`**set**
Read-only reference LilyPond context setting component plug-in.

`Cluster.`**spanners**
Read-only reference to unordered set of spanners attached to component.

`Cluster.storage_format`
Storage format of Abjad object.

Return string.

`Cluster.timespan`
Read-only timespan of component.

`Cluster.timespan_in_seconds`
Read-only timespan of component in seconds.

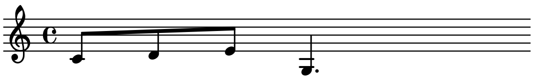
Read/write properties

`Cluster.is_parallel`
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

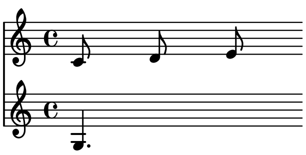
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

Methods

`Cluster.`**`append`**`(`*component*`)`

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

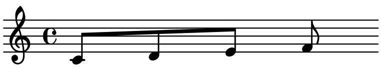
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`Cluster.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

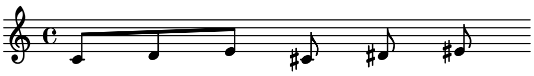
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Cluster.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c’8 d’8 e’8")
```

```
>>> note = container[-1]
>>> note
Note("e’8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Cluster.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

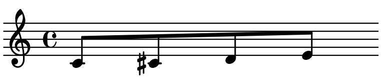
```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs’8"))
```

```
>>> f(container)
{
    c’8 [
    cs’8
    d’8
    e’8 ]
}
```

```
>>> show(container)
```



Return none.

`Cluster.`**`pop`**(*i=-1*)

Pop component at index *i* from container:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

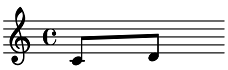
```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Cluster.`**`remove`**(*component*)

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

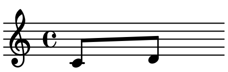
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Cluster.`**`__add__`**(*expr*)

Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

`Cluster.__contains__(expr)`
True if expr is in container, otherwise False.

`Cluster.__copy__(*args)`

`Cluster.__delitem__(i)`
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

`Cluster.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Cluster.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Cluster.__getitem__(i)`
Return component at index i in container. Shallow traversal of container for numeric indices only.

`Cluster.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Cluster.__iadd__(expr)`
__iadd__ avoids unnecessary copying of structures.

`Cluster.__imul__(total)`
Multiply contents of container 'total' times. Return multiplied container.

`Cluster.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Cluster.__len__()`
Return nonnegative integer number of components in container.

`Cluster.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Cluster.__mul__(n)`

`Cluster.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

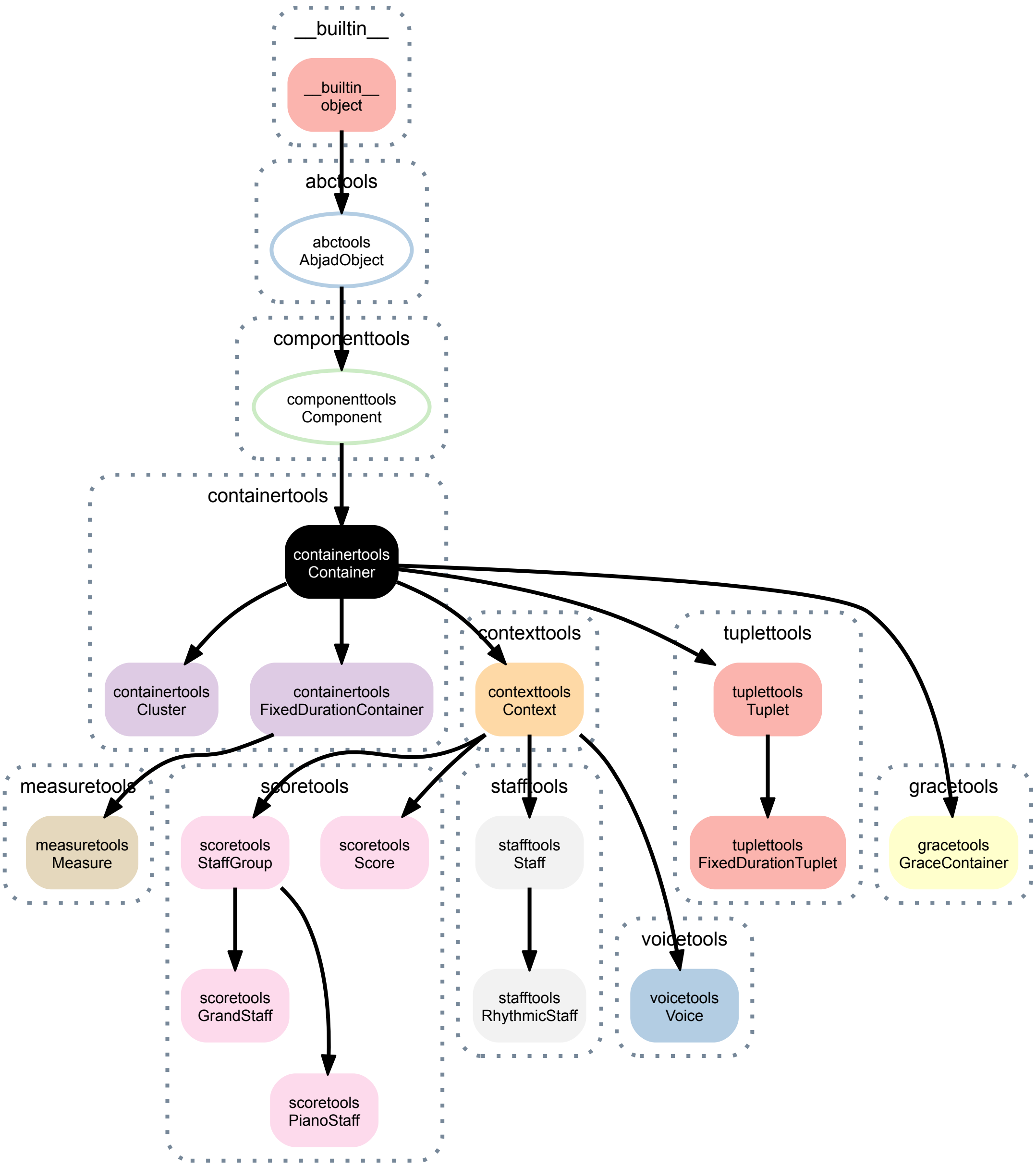
`Cluster.__radd__(expr)`
Extend container by contents of expr to the right.

`Cluster.__repr__()`

`Cluster.__rmul__(n)`

`Cluster.__setitem__(i, expr)`
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

4.1.2 containertools.Container



class `containertools.`**`Container`**(*music=None*, ***kwargs*)

Abjad model of a music container:

```
>>> container = Container("c'8 d'8 e'8 f'8")
>>> f(container)
{
    c'8
    d'8
    e'8
    f'8
}
```

Return Container instance.

Read-only properties

`Container.`**`contents_duration`**

`Container.`**`descendants`**
Read-only reference to component descendants score selection.

`Container.`**`duration`**

`Container.`**`duration_in_seconds`**

`Container.`**`leaves`**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`Container.`**`lilypond_format`**

`Container.`**`lineage`**
Read-only reference to component lineage score selection.

`Container.`**`music`**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`Container.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Container.`**`parent`**

`Container.`**`parentage`**
Read-only reference to component parentage score selection.

`Container.`**`preprolated_duration`**

`Container.`**`prolation`**

`Container.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Container.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`Container.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Container.`**`timespan`**
Read-only timespan of component.

`Container.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

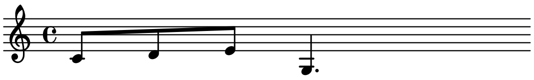
Read/write properties

`Container.`**`is_parallel`**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

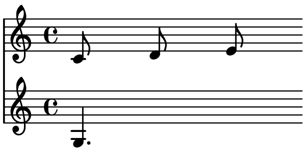
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

Methods

`Container.`**append**(*component*)

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

`Container.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

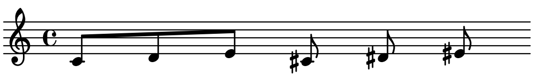
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Container.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Container.`**`insert`**`(`*i*, *component*`)`

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

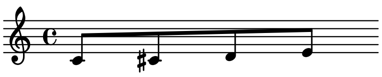
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Container.`**`pop`**`(`*i=-1*`)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

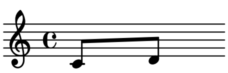
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Container.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

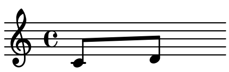
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Container.`**`__add__`**`(`*expr*`)`

Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

`Container.`**`__contains__`**`(`*expr*`)`

True if expr is in container, otherwise False.

`Container.`**`__copy__`**`(`**args*`)`

`Container.`**`__delitem__`**`(`*i*`)`

Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

`Container.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`Container.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`Container.`**`__getitem__`**`(`*i*`)`

Return component at index i in container. Shallow traversal of container for numeric indices only.

`Container.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Container.__iadd__(expr)`
__iadd__ avoids unnecessary copying of structures.

`Container.__imul__(total)`
Multiply contents of container 'total' times. Return multiplied container.

`Container.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Container.__len__()`
Return nonnegative integer number of components in container.

`Container.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Container.__mul__(n)`

`Container.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

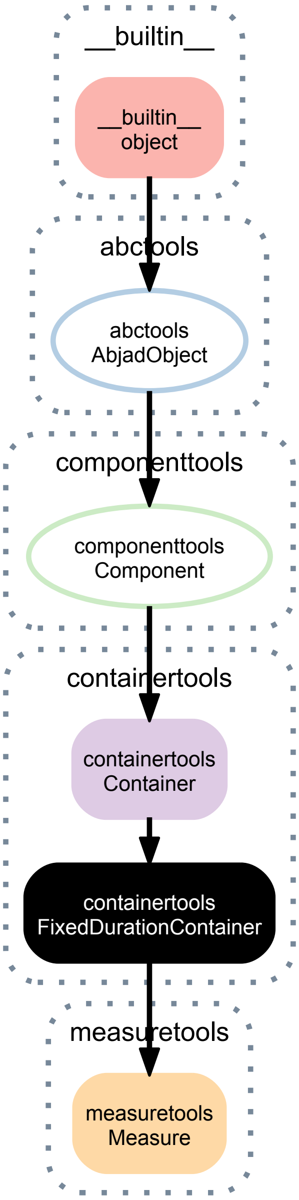
`Container.__radd__(expr)`
Extend container by contents of expr to the right.

`Container.__repr__()`
String format of container for interpreter display.

`Container.__rmul__(n)`

`Container.__setitem__(i, expr)`
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

4.1.3 containertools.FixedDurationContainer



class `containertools.`**`FixedDurationContainer`**(*target_duration*, *music=None*, ***kwargs*)

New in version 2.9. Fixed-duration container:

```
>>> container = containertools.FixedDurationContainer((3, 8), "c'8 d'8 e'8")
```

```
>>> container
FixedDurationContainer(Duration(3, 8), [Note("c'8"), Note("d'8"), Note("e'8")])
```

```
>>> f(container)
{
    c'8
    d'8
    e'8
}
```

Fixed-duration containers extend container behavior with format-time checking against a user-specified target duration.

Return fixed-duration container.

Read-only properties

`FixedDurationContainer.`**contents_duration**

`FixedDurationContainer.`**descendants**
Read-only reference to component descendants score selection.

`FixedDurationContainer.`**duration**

`FixedDurationContainer.`**duration_in_seconds**

`FixedDurationContainer.`**is_full**
True when preprolated duration equals target duration.

`FixedDurationContainer.`**is_misfilled**
True when preprolated duration does not equal target duration.

`FixedDurationContainer.`**is_overfull**
True when preprolated duration is greater than target duration.

`FixedDurationContainer.`**is_underfull**
True when preprolated duration is less than target duration.

`FixedDurationContainer.`**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`FixedDurationContainer.`**lilypond_format**
Read-only LilyPond format of fixed-duration container.

`FixedDurationContainer.`**lineage**
Read-only reference to component lineage score selection.

`FixedDurationContainer.`**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`FixedDurationContainer.`**override**
Read-only reference to LilyPond grob override component plug-in.

`FixedDurationContainer.`**parent**

`FixedDurationContainer.`**parentage**
Read-only reference to component parentage score selection.

`FixedDurationContainer.`**preprolated_duration**

`FixedDurationContainer.`**prolation**

`FixedDurationContainer.`**set**
Read-only reference LilyPond context setting component plug-in.

`FixedDurationContainer.`**spanners**
Read-only reference to unordered set of spanners attached to component.

`FixedDurationContainer.`**storage_format**
Storage format of Abjad object.

Return string.

FixedDurationContainer.**timespan**
Read-only timespan of component.

FixedDurationContainer.**timespan_in_seconds**
Read-only timespan of component in seconds.

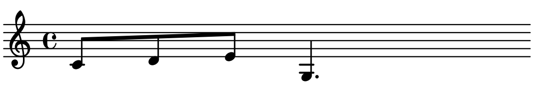
Read/write properties

FixedDurationContainer.**is_parallel**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```

Return none.

FixedDurationContainer.**target_duration**
Read / write target duration of fixed-duration container.

Methods

`FixedDurationContainer.`**append**(*component*)

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

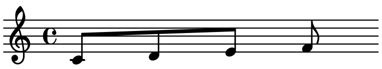
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`FixedDurationContainer.`**extend**(*expr*)

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

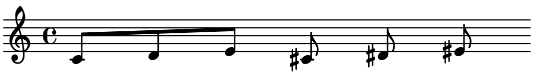
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`FixedDurationContainer.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`FixedDurationContainer.`**`insert`**`(`*i*, *component*`)`

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

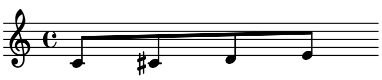
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`FixedDurationContainer.`**`pop`**`(`*i=-1*`)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

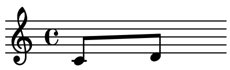
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

FixedDurationContainer.**remove**(*component*)
Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

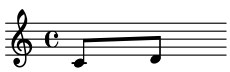
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

FixedDurationContainer.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

FixedDurationContainer.**__contains__**(*expr*)
True if expr is in container, otherwise False.

FixedDurationContainer.**__copy__**(**args*)

FixedDurationContainer.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

FixedDurationContainer.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

FixedDurationContainer.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationContainer.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

FixedDurationContainer.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

FixedDurationContainer.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

FixedDurationContainer.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

FixedDurationContainer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationContainer.**__len__**()
Return nonnegative integer number of components in container.

FixedDurationContainer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationContainer.**__mul__**(*n*)

FixedDurationContainer.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

FixedDurationContainer.**__radd__**(*expr*)
Extend container by contents of expr to the right.

FixedDurationContainer.**__repr__**()

FixedDurationContainer.**__rmul__**(*n*)

FixedDurationContainer.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

4.2 Functions

4.2.1 containertools.all_are_containers

`containertools.all_are_containers(expr)`

New in version 2.6. True when *expr* is a sequence of Abjad containers:

```
>>> containers = 3 * Container("c'8 d'8 e'8")
```

```
>>> containertools.all_are_containers(containers)
True
```

True when *expr* is an empty sequence:

```
>>> containertools.all_are_containers([])
True
```

Otherwise false:

```
>>> containertools.all_are_containers('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

4.2.2 containertools.delete_contents_of_container

`containertools.delete_contents_of_container(container)`

Delete contents of *container*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> containertools.delete_contents_of_container(staff)
Selection(Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"))
```

```
>>> f(staff)
\new Staff {
}
```

Return *container* contents.

4.2.3 containertools.delete_contents_of_container_starting_at_or_after_offset

`containertools.delete_contents_of_container_starting_at_or_after_offset(container, prolated_offset)`

New in version 2.0. Delete contents of *container* starting at or after *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> containertools.delete_contents_of_container_starting_at_or_after_offset(
...     staff, Duration(1, 8))
Staff{1}
```

```
>>> f(staff)
\new Staff {
    c'8 [ ]
}
```

Return *container*.

4.2.4 containertools.delete_contents_of_container_starting_before_or_at_offset

containertools.**delete_contents_of_container_starting_before_or_at_offset**(*container*, *prolated_offset*)

New in version 2.0. Delete contents of *container* starting before or at *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> containertools.delete_contents_of_container_starting_before_or_at_offset(
...     staff, Duration(1, 8))
Staff{2}
```

```
>>> f(staff)
\new Staff {
    e'8 [
    f'8 ]
}
```

Return *container*.

4.2.5 containertools.delete_contents_of_container_starting_strictly_after_offset

containertools.**delete_contents_of_container_starting_strictly_after_offset**(*container*, *prolated_offset*)

New in version 2.0. Delete contents of *container* starting strictly after *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
```

```
        e'8
        f'8 ]
    }
```

```
    >>> containertools.delete_contents_of_container_starting_strictly_after_offset(
    ...     staff, Duration(1, 8))
    Staff{2}
```

```
    >>> f(staff)
    \new Staff {
        c'8 [
        d'8 ]
    }
```

Return *container*.

4.2.6 containertools.delete_contents_of_container_starting_strictly_before_offset

containertools.**delete_contents_of_container_starting_strictly_before_offset**(*container*, *prolated_offset*)

New in version 2.0. Delete contents of *container* contents starting strictly before *prolated_offset*:

```
    >>> staff = Staff("c'8 d'8 e'8 f'8")
    >>> beamtools.BeamSpanner(staff.leaves)
    BeamSpanner(c'8, d'8, e'8, f'8)
```

```
    >>> f(staff)
    \new Staff {
        c'8 [
        d'8
        e'8
        f'8 ]
    }
```

```
    >>> containertools.delete_contents_of_container_starting_strictly_before_offset(
    ...     staff, Duration(1, 8))
    Staff{3}
```

```
    >>> f(staff)
    \new Staff {
        d'8 [
        e'8
        f'8 ]
    }
```

Return *container*.

4.2.7 containertools.eject_contents_of_container

containertools.**eject_contents_of_container**(*container*)

New in version 2.0. Eject contents of *container*:

```
    >>> container = Container("c'8 d'8 e'8 f'8")
```

```
    >>> f(container)
    {
        c'8
        d'8
        e'8
        f'8
    }
```

```
    >>> containertools.eject_contents_of_container(container)
    Selection(Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"))
```

```
>>> container
{}
```

```
>>> f(container)
{
}
```

Return list of *container* contents.

4.2.8 containertools.fuse_like_named_contiguous_containers_in_expr

containertools.**fuse_like_named_contiguous_containers_in_expr**(*expr*)

Fuse like-named contiguous containers in *expr*:

```
>>> staff = Staff(r"\new Voice { c'8 d'8 } \new Voice { e'8 f'8 }")
>>> staff[0].name = 'soprano'
>>> staff[1].name = 'soprano'
```

```
>>> f(staff)
\new Staff {
    \context Voice = "soprano" {
        c'8
        d'8
    }
    \context Voice = "soprano" {
        e'8
        f'8
    }
}
```

```
>>> containertools.fuse_like_named_contiguous_containers_in_expr(staff)
Staff{1}
```

```
>>> f(staff)
\new Staff {
    \context Voice = "soprano" {
        c'8
        d'8
        e'8
        f'8
    }
}
```

Return *expr*.

4.2.9 containertools.get_element_starting_at_exactly_offset

containertools.**get_element_starting_at_exactly_offset**(*container*, *prolated_offset*)

New in version 2.0. Get *container* element starting at exactly *prolated_offset*:

```
>>> voice = Voice("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> containertools.get_element_starting_at_exactly_offset(voice, Duration(6, 8))
Note("b'8")
```

Raise missing component error when no *container* element starts at exactly *prolated_offset*.

4.2.10 containertools.get_first_container_in_improper_parentage_of_component

containertools.**get_first_container_in_improper_parentage_of_component**(*component*)

New in version 2.0. Get first container in improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> containertools.get_first_container_in_improper_parentage_of_component(staff[1])
Staff{4}
```

Return container or none.

4.2.11 containertools.get_first_container_in_proper_parentage_of_component

containertools.**get_first_container_in_proper_parentage_of_component**(*component*)
New in version 2.0. Get first container in proper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> containertools.get_first_container_in_proper_parentage_of_component(staff[1])
Staff{4}
```

Return container or none.

4.2.12 containertools.get_first_element_starting_at_or_after_offset

containertools.**get_first_element_starting_at_or_after_offset**(*container*, *prolated_offset*)
New in version 2.0. Get first *container* element starting at or after *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> containertools.get_first_element_starting_at_or_after_offset(staff, Duration(1, 8))
Note("d'8")
```

Return component.

Return none when no *container* element starts at or after *prolated_offset*.

4.2.13 containertools.get_first_element_starting_before_or_at_offset

containertools.**get_first_element_starting_before_or_at_offset**(*container*, *prolated_offset*)
New in version 2.0. Get first *container* element starting before or at *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> containertools.get_first_element_starting_before_or_at_offset(staff, Duration(1, 8))
Note("d'8")
```

Return component.

Return none when no *container* element starts before or at *prolated_offset*.

4.2.14 containertools.get_first_element_starting_strictly_after_offset

containertools.**get_first_element_starting_strictly_after_offset**(*container*, *prolated_offset*)

New in version 2.0. Get first *container* element starting strictly after *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> containertools.get_first_element_starting_strictly_after_offset(staff, Duration(1, 8))
Note("e'8")
```

Return component.

Return none when no *container* element starts strictly after *prolated_offset*.

4.2.15 containertools.get_first_element_starting_strictly_before_offset

containertools.**get_first_element_starting_strictly_before_offset**(*container*, *prolated_offset*)

New in version 2.0. Get first *container* element starting strictly before *prolated_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> containertools.get_first_element_starting_strictly_before_offset(staff, Duration(1, 8))
Note("c'8")
```

Return component.

Return none when *container* element starts stirctly before *prolated_offset*.

4.2.16 containertools.insert_component

containertools.**insert_component**(*container*, *i*, *component*, *fracture_spanners=False*)

New in version 2.10. Insert *component* into *container* at index *i*.

Example 1. Insert *component* into *container* at index *i*. Do not fracture crossing spanners:

```
>>> staff = Staff("c'8 [ d'8 e'8 f'8 ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> containertools.insert_component(staff, 1, Note("cs'8"), fracture_spanners=False)
Staff{5}
```

```
>>> f(staff)
\new Staff {
    c'8 [
    cs'8
    d'8
    e'8
    f'8 ]
}
```

Example 2. Insert *component* into *container* at index *i*. Fracture crossing spanners.

```
>>> staff = Staff("c'8 [ d'8 e'8 f'8 ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> parts = containertools.insert_component(staff, 1, Rest('r8'), fracture_spanners=True)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ]
    r8
    d'8 [
    e'8
    f'8 ]
}
```

Return *container* or list of fractured spanners.

4.2.17 containertools.move_parentage_children_and_spanners_from_components_to_empt

containertools.**move_parentage_children_and_spanners_from_components_to_empty_container**(*c*
c
t

Move parentage, children and spanners from donor *components* to recipient empty *container*:

```
>>> voice = Voice("{ c'8 [ d'8 } { e'8 f'8 } { g'8 a'8 ] }")
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8 ]
    }
}
```

```
>>> tuplet = Tuplet(Fraction(3, 4), [])
>>> containertools.move_parentage_children_and_spanners_from_components_to_empty_container(
... voice[:2], tuplet)
```

```
>>> f(voice)
\new Voice {
    \fraction \times 3/4 {
        c'8 [
        d'8
        e'8
        f'8
    }
    {
        g'8
        a'8 ]
    }
}
```

Return none.

4.2.18 containertools.remove_leafless_containers_in_expr

containertools.**remove_leafless_containers_in_expr**(*expr*)

Remove empty containers in *expr*:

```
>>> staff = Staff("{ c'8 d'8 } { e'8 f'8 } { g'8 a'8 } { b'8 c''8 }")
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner({c'8, d'8}, {e'8, f'8}, {g'8, a'8}, {b'8, c''8})
```

```
>>> containertools.delete_contents_of_container(staff[1])
Selection(Note("e'8"), Note("f'8"))
>>> containertools.delete_contents_of_container(staff[-1])
Selection(Note("b'8"), Note("c''8"))
```

```
>>> f(staff)
\new Staff {
    {
        c'8 [
        d'8
    }
    {
    }
    {
        g'8
        a'8 ]
    }
    {
    }
}
```

```
>>> containertools.remove_leafless_containers_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        c'8 [
        d'8
    }
    {
        g'8
        a'8 ]
    }
}
```

Return none.

4.2.19 containertools.repeat_contents_of_container

containertools.**repeat_contents_of_container**(*container*, *total=2*)

New in version 1.1. Repeat contents of *container*:

```
>>> staff = Staff("c'8 d'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
}
```

```
>>> containertools.repeat_contents_of_container(staff, 3)
Staff{6}
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
```

```
        c'8 [
        d'8 ]
        c'8 [
        d'8 ]
    }
```

Leave *container* unchanged when *total* is `1`.

Empty *container* when *total* is `0`.

Return *container*.

4.2.20 containertools.repeat_last_n_elements_of_container

containertools.**repeat_last_n_elements_of_container**(*container*, *n=1*, *total=2*)

New in version 1.1. Repeat last *n* elements of *container*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> containertools.repeat_last_n_elements_of_container(staff, n=2, total=3)
Staff{8}
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
    e'8 [
    f'8 ]
    e'8 [
    f'8 ]
}
```

Return *container*.

4.2.21 containertools.replace_container_slice_with_rests

containertools.**replace_container_slice_with_rests**(*container*, *start=None*, *stop=None*, *decrease_durations_monotonically=True*)

New in version 2.10. Replace *container* slice from *start* to *stop* with rests that decrease durations monotonically.

Example 1. Replace all container elements:

```
>>> container = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b' c''8")
```

```
>>> container = containertools.replace_container_slice_with_rests(container)

>>> f(container)
\new Staff {
    r1
}
```

Example 2. Replace container elements from `1` forward:

```
>>> container = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b' c''8")
```

```
>>> container = containertools.replace_container_slice_with_rests(
...     container, start=1)
```

```
>>> f(container)
\new Staff {
    c'8
    r2..
}
```

Example 3. Replace container elements from `1` to `2`:

```
>>> container = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b' c''8")
```

::

```
>>> container= containertools.replace_container_slice_with_rests(
...         container, start=1, stop=2)
```

```
>>> f(container)
\new Staff {
    c'8
    r8
    e'8
    f'8
    g'8
    a'8
    b'8
    c''8
}
```

Return *container*.

4.2.22 containertools.replace_contents_of_target_container_with_contents_of_source_cont

containertools.**replace_contents_of_target_container_with_contents_of_source_container**(*ta so*

New in version 2.0. Replace contents of *target_container* with contents of *source_container*:

```
>>> staff = Staff(Tuplet(Fraction(2, 3), "c'8 d'8 e'8") * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(
...     staff)
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, ... [5] ..., c''8, d''8)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
    \times 2/3 {
        f'8
        g'8
        a'8
    }
    \times 2/3 {
        b'8
        c''8
        d''8 ]
    }
}
```

```
>>> container = Container("c'8 d'8 e'8")
>>> spannertools.SlurSpanner(container.leaves)
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(container)
{
    c'8 (
    d'8
    e'8 )
}
```

```
>>> containertools.replace_contents_of_target_container_with_contents_of_source_container(
...     staff[1], container)
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
    \times 2/3 {
        c'8 (
        d'8
        e'8 )
    }
    \times 2/3 {
        b'8
        c''8
        d''8 ]
    }
}
```

Leave *source_container* empty:

```
>>> container
{}
```

Return *target_container*.

4.2.23 containertools.report_container_modifications

containertools.**report_container_modifications**(*container*)

Report *container* modifications as string:

```
>>> container = Container("c'8 d'8 e'8 f'8")
>>> container.override.note_head.color = 'red'
>>> container.override.note_head.style = 'harmonic'
```

```
>>> f(container)
{
    \override NoteHead #'color = #red
    \override NoteHead #'style = #'harmonic
    c'8
    d'8
    e'8
    f'8
    \revert NoteHead #'color
    \revert NoteHead #'style
}
```

```
>>> string = containertools.report_container_modifications(container)
```

```
>>> print string
{
    \override NoteHead #'color = #red
    \override NoteHead #'style = #'harmonic
```

```
        %%% 4 components omitted %%%
        \revert NoteHead #'color
        \revert NoteHead #'style
    }
```

Return string.

4.2.24 containertools.reverse_contents_of_container

containertools.**reverse_contents_of_container**(*container*)

New in version 1.1. Reverse contents of *container*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves[:2])
BeamSpanner(c'8, d'8)
>>> spannertools.SlurSpanner(staff.leaves[2:])
SlurSpanner(e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
    e'8 (
    f'8 )
}
```

```
>>> containertools.reverse_contents_of_container(staff)
Staff{4}
```

```
>>> f(staff)
\new Staff {
    f'8 (
    e'8 )
    d'8 [
    c'8 ]
}
```

Return *container*.

4.2.25 containertools.scale_contents_of_container

containertools.**scale_contents_of_container**(*container*, *multiplier*)

New in version 1.1. Scale contents of *container* by dot *multiplier*:

```
>>> staff = Staff("c'8 d'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
}
```

```
>>> containertools.scale_contents_of_container(staff, Duration(3, 2))
Staff{2}
```

```
>>> f(staff)
\new Staff {
    c'8. [
    d'8. ]
}
```

Scale contents of *container* by tie *multiplier*:

```
>>> staff = Staff("c'8 d'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
}
```

```
>>> containertools.scale_contents_of_container(staff, Duration(5, 4))
Staff{4}
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'32
    d'8 ~
    d'32 ]
}
```

Scale contents of *container* by *multiplier* without power-of-two denominator:

```
>>> staff = Staff("c'8 d'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ]
}
```

```
>>> containertools.scale_contents_of_container(staff, Duration(4, 3))
Staff{2}
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'4 [
    }
    \times 2/3 {
        d'4 ]
    }
}
```

Return *container*.

4.2.26 containertools.set_container_multiplier

containertools.**set_container_multiplier**(*container*, *multiplier*)

Set *container multiplier*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> containertools.set_container_multiplier(tuplet, Multiplier(3, 4))
```

```
>>> f(tuplet)
\fraction \times 3/4 {
    c'8
```

```
        d'8
        e'8
    }
```

Return none.

4.2.27 containertools.split_container_at_index

containertools.**split_container_at_index**(*component, i, fracture_spanners=False*)

New in version 1.1. General component index split algorithm. Works on leaves, tuplets, measures, contexts and unqualified containers. Keyword controls spanner behavior at split time. Use containertools.split_container_at_index_and_fracture_crossing_spanners() to fracture spanners. Use containertools.split_container_at_index_and_do_not_fracture_crossing_spanners() to leave spanners unchanged.

Example 1. Split container and do not fracture crossing spanners:

```
>>> voice = Voice(Measure((3, 8), "c'8 c'8 c'8") * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice[:])
```

```
>>> f(voice)
\new Voice {
    {
        \time 3/8
        c'8 [
        d'8
        e'8
    }
    {
        f'8
        g'8
        a'8 ]
    }
}
```

```
>>> containertools.split_container_at_index(voice[1], 1, fracture_spanners=False)
(Measure(1/8, [f'8]), Measure(2/8, [g'8, a'8]))
```

```
>>> f(voice)
\new Voice {
    {
        \time 3/8
        c'8 [
        d'8
        e'8
    }
    {
        \time 1/8
        f'8
    }
    {
        \time 2/8
        g'8
        a'8 ]
    }
}
```

Example 2. Split container and fracture crossing spanners:

```
>>> voice = Voice(tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 c'8 c'8") * 2)
>>> tuplet = voice[1]
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice[:])
```

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        c'8 [
```

```
            d'8
            e'8
        }
        \times 2/3 {
            f'8
            g'8
            a'8 ]
        }
    }
```

```
>>> left, right = containertools.split_container_at_index(
...         tuplet, 1, fracture_spanners=True)
```

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
    \times 2/3 {
        f'8 ]
    }
    \times 2/3 {
        g'8 [
        a'8 ]
    }
}
```

Leave spanners and leaves untouched.

Resize resizable containers.

Preserve container multiplier.

Preserve time signature denominator.

Return split parts.

4.2.28 containertools.split_container_by_counts

`containertools.`**`split_container_by_counts`**(*components*, *counts*, *fracture_spanners=False*, *cyclic=False*)

New in version 1.1. Partition Python list of zero or more Abjad components. Partition by zero or more positive integers in counts list. Fracture spanners or not according to keyword. Read counts in list cyclically or not according to keyword. Return list of component lists.

Example 1. Split container cyclically by counts and do not fracture crossing spanners:

```
>>> container = Container("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> voice = Voice([container])
>>> beam = beamtools.BeamSpanner(voice)
>>> slur = spannertools.SlurSpanner(container)
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
        d'8
        e'8
        f'8
        g'8
        a'8
        b'8
        c''8 ] )
    }
}
```

```
>>> containertools.split_container_by_counts(
...     container, [1, 3], cyclic=True, fracture_spanners=False)
[[{c'8}], [{d'8, e'8, f'8}], [{g'8}], [{a'8, b'8, c''8}]]
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
    }
    {
        d'8
        e'8
        f'8
    }
    {
        g'8
    }
    {
        a'8
        b'8
        c''8 ] )
    }
}
```

Example 2. Split container cyclically by counts and fracture crossing spanners:

```
>>> container = Container("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> voice = Voice([container])
>>> beam = beamtools.BeamSpanner(voice)
>>> slur = spannertools.SlurSpanner(container)
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
        d'8
        e'8
        f'8
        g'8
        a'8
        b'8
        c''8 ] )
    }
}
```

```
>>> containertools.split_container_by_counts(
...     container, [1, 3], cyclic=True, fracture_spanners=True)
[[{c'8}], [{d'8, e'8, f'8}], [{g'8}], [{a'8, b'8, c''8}]]
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ ( )
    }
    {
        d'8 (
        e'8
        f'8 )
    }
    {
        g'8 ( )
    }
    {
        a'8 (
        b'8
        c''8 ] )
    }
}
```

Example 3. Split container once by counts and do not fracture crossing spanners:

```
>>> container = Container("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> voice = Voice([container])
>>> beam = beamtools.BeamSpanner(voice)
>>> slur = spannertools.SlurSpanner(container)
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
        d'8
        e'8
        f'8
        g'8
        a'8
        b'8
        c''8 ] )
    }
}
```

```
>>> containertools.split_container_by_counts(
...     container, [1, 3], cyclic=False, fracture_spanners=False)
[[{c'8}], [{d'8, e'8, f'8}], [{g'8, a'8, b'8, c''8}]]
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
    }
    {
        d'8
        e'8
        f'8
    }
    {
        g'8
        a'8
        b'8
        c''8 ] )
    }
}
```

Example 4. Split container once by counts and fracture crossing spanners:

```
>>> container = Container("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> voice = Voice([container])
>>> beam = beamtools.BeamSpanner(voice)
>>> slur = spannertools.SlurSpanner(container)
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ (
        d'8
        e'8
        f'8
        g'8
        a'8
        b'8
        c''8 ] )
    }
}
```

```
>>> containertools.split_container_by_counts(
...     container, [1, 3], cyclic=False, fracture_spanners=True)
[[{c'8}], [{d'8, e'8, f'8}], [{g'8, a'8, b'8, c''8}]]
```

```
>>> f(voice)
\new Voice {
    {
        c'8 [ ( )
    }
```

```
        {
            d'8 (
            e'8
            f'8 )
        }
        {
            g'8 (
            a'8
            b'8
            c''8 ] )
        }
    }
```

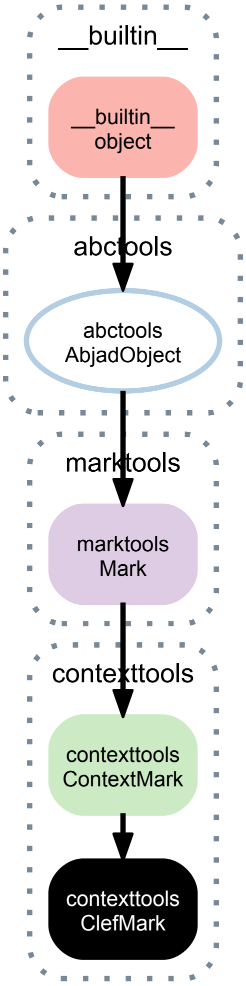
Return list of split parts.

CHAPTER FIVE

CONTEXTTOOLS

5.1 Concrete Classes

5.1.1 contexttools.ClefMark



class `contexttools.ClefMark`(*arg*, *target_context=None*)

New in version 2.0. Abjad model of a clef:

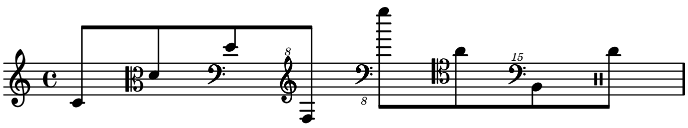
```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{8})
```

```
>>> clef = contexttools.ClefMark('alto')(staff[1])
>>> clef = contexttools.ClefMark('bass')(staff[2])
>>> clef = contexttools.ClefMark('treble^8')(staff[3])
>>> clef = contexttools.ClefMark('bass_8')(staff[4])
>>> clef = contexttools.ClefMark('tenor')(staff[5])
>>> clef = contexttools.ClefMark('bass^15')(staff[6])
>>> clef = contexttools.ClefMark('percussion')(staff[7])
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    \clef "alto"
    d'8
    \clef "bass"
    e'8
    \clef "treble^8"
    f'8
    \clef "bass_8"
    g'8
    \clef "tenor"
    a'8
    \clef "bass^15"
    b'8
    \clef "percussion"
    c''8
}
```

```
>>> show(staff)
```



Clef marks target the staff context by default.

Read-only properties

`ClefMark.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`ClefMark.`**`lilypond_format`**

Read-only LilyPond format of clef:

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.lilypond_format
'\\clef "treble"'
```

Return string.

`ClefMark.`**`middle_c_position`**

Read-only middle-C position of clef:

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.middle_c_position
-6
```

Return integer number of stafflines.

`ClefMark.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`ClefMark.`**`storage_format`**

Storage format of Abjad object.

Return string.

`ClefMark.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`ClefMark.`**`clef_name`**

Get clef name:

```
>>> clef = contexttools.ClefMark('treble')
>>> clef.clef_name
'treble'
```

Set clef name:

```
>>> clef.clef_name = 'alto'
>>> clef.clef_name
'alto'
```

Return string.

Methods

`ClefMark.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`ClefMark.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

ClefMark.**__call__**(*args*)

ClefMark.**__copy__**(*args*)

ClefMark.**__deepcopy__**(*args*)

ClefMark.**__delattr__**(*args*)

ClefMark.**__eq__**(*arg*)

ClefMark.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ClefMark.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ClefMark.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

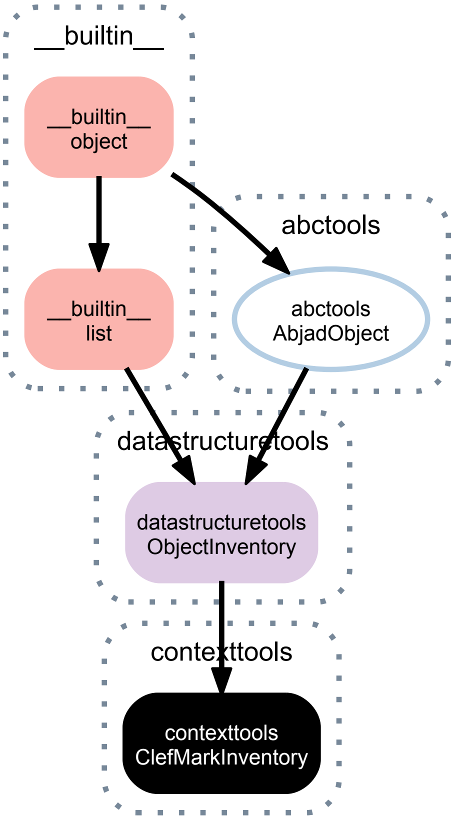
ClefMark.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ClefMark.**__ne__**(*arg*)

ClefMark.**__repr__**()

5.1.2 contexttools.ClefMarkInventory



class `contexttools.ClefMarkInventory` (*tokens=None*, *name=None*)

New in version 2.8. Abjad model of an ordered list of clefs:

```
>>> inventory = contexttools.ClefMarkInventory(['treble', 'bass'])
```

```
>>> inventory
ClefMarkInventory([ClefMark('treble'), ClefMark('bass')])
```

```
>>> 'treble' in inventory
True
```

```
>>> contexttools.ClefMark('treble') in inventory
True
```

```
>>> 'alto' in inventory
False
```

Clef mark inventories implement list interface and are mutable.

Read-only properties

`ClefMarkInventory.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`ClefMarkInventory.`**`name`**

Read / write name of inventory.

Methods

`ClefMarkInventory.`**`append`** (*token*)

Change *token* to item and append.

`ClefMarkInventory.`**`count`** (*value*) → integer – return number of occurrences of value

`ClefMarkInventory.`**`extend`** (*tokens*)

Change *tokens* to items and extend.

`ClefMarkInventory.`**`index`** (*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

`ClefMarkInventory.`**`insert`** ()

L.insert(index, object) – insert object before index

`ClefMarkInventory.`**`pop`** ([*index*]) → item – remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

`ClefMarkInventory.`**`remove`** ()

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`ClefMarkInventory.`**`reverse`** ()

L.reverse() – reverse *IN PLACE*

`ClefMarkInventory.`**`sort`** ()

L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`ClefMarkInventory.__add__()`
x.__add__(y) <==> x+y

`ClefMarkInventory.__contains__(token)`

`ClefMarkInventory.__delitem__()`
x.__delitem__(y) <==> del x[y]

`ClefMarkInventory.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`ClefMarkInventory.__eq__()`
x.__eq__(y) <==> x==y

`ClefMarkInventory.__ge__()`
x.__ge__(y) <==> x>=y

`ClefMarkInventory.__getitem__()`
x.__getitem__(y) <==> x[y]

`ClefMarkInventory.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ClefMarkInventory.__gt__()`
x.__gt__(y) <==> x>y

`ClefMarkInventory.__iadd__()`
x.__iadd__(y) <==> x+=y

`ClefMarkInventory.__imul__()`
x.__imul__(y) <==> x*=y

`ClefMarkInventory.__iter__() <==> iter(x)`

`ClefMarkInventory.__le__()`
x.__le__(y) <==> x<=y

`ClefMarkInventory.__len__() <==> len(x)`

`ClefMarkInventory.__lt__()`
x.__lt__(y) <==> x<y

`ClefMarkInventory.__mul__()`
x.__mul__(n) <==> x*n

`ClefMarkInventory.__ne__()`
x.__ne__(y) <==> x!=y

`ClefMarkInventory.__repr__()`

`ClefMarkInventory.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

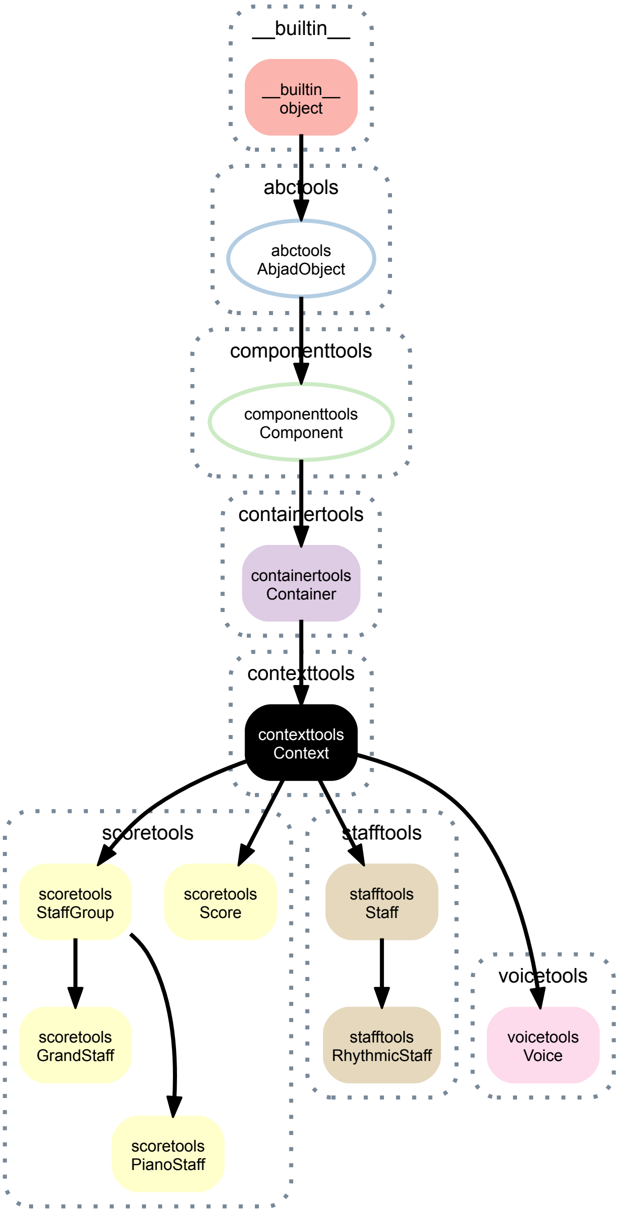
`ClefMarkInventory.__rmul__()`
x.__rmul__(n) <==> n*x

`ClefMarkInventory.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`ClefMarkInventory.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

5.1.3 contexttools.Context



class `contexttools.`**`Context`** (*music=None*, *context_name='Context'*, *name=None*)

New in version 1.0. Abjad model of a horizontal layer of music.

```
>>> context = contexttools.Context(
... name='MeterVoice', context_name='TimeSignatureContext')
```

```
>>> context
TimeSignatureContext-"MeterVoice"{}
```

```
>>> f(context)
\context TimeSignatureContext = "MeterVoice" {
}
```

Return context object.

Read-only properties

`Context.`**`contents_duration`**

`Context.`**`descendants`**
Read-only reference to component descendants score selection.

`Context.`**`duration`**

`Context.`**`duration_in_seconds`**

`Context.`**`engraver_consists`**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

`Context.`**`engraver_removals`**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

`Context.`**`is_semantic`**

`Context.`**`leaves`**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`Context.`**`lilypond_format`**

`Context.`**`lineage`**
Read-only reference to component lineage score selection.

`Context.`**`music`**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`Context.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Context.`**`parent`**

`Context.`**`parentage`**
Read-only reference to component parentage score selection.

`Context.`**`preprolated_duration`**

`Context.`**`prolation`**

`Context.`**set**

Read-only reference LilyPond context setting component plug-in.

`Context.`**spanners**

Read-only reference to unordered set of spanners attached to component.

`Context.`**storage_format**

Storage format of Abjad object.

Return string.

`Context.`**timespan**

Read-only timespan of component.

`Context.`**timespan_in_seconds**

Read-only timespan of component in seconds.

Read/write properties

`Context.`**context_name**

Read / write name of context as a string.

`Context.`**is_nonsemantic**

Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

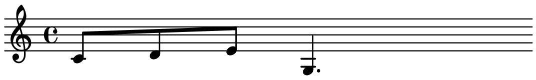
`Context.`**is_parallel**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

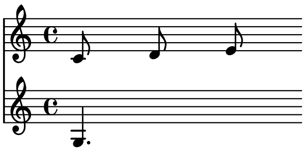
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

`Context.`**`name`**
Read-write name of context. Must be string or none.

Methods

`Context.`**`append`**`(`*component*`)`
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

`Context.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

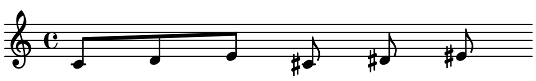
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Context.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Context.insert(i, component)`
Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

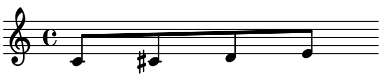
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Context.pop(i=-1)`
Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

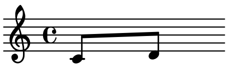
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Context.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

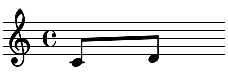
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Context.`**`__add__`**`(`*expr*`)`

Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

`Context.`**`__contains__`**`(`*expr*`)`

True if expr is in container, otherwise False.

`Context.`**`__copy__`**`(`**args*`)`

`Context.`**`__delitem__`**`(`*i*`)`

Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

`Context.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

Context.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Context.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

Context.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Context.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

Context.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

Context.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Context.**__len__**()
Return nonnegative integer number of components in container.

Context.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Context.**__mul__**(*n*)

Context.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

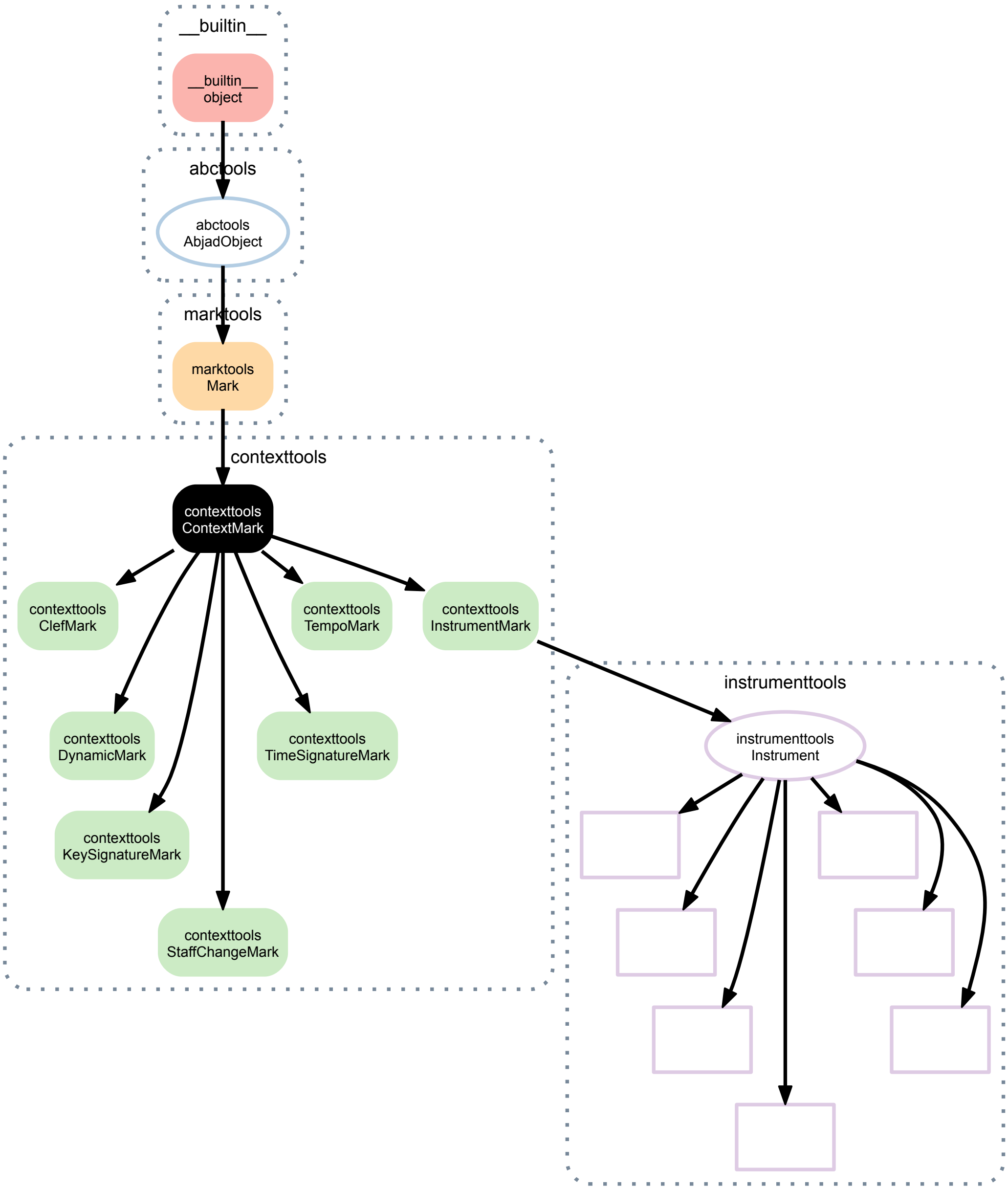
Context.**__radd__**(*expr*)
Extend container by contents of expr to the right.

Context.**__repr__**()

Context.**__rmul__**(*n*)

Context.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

5.1.4 contexttools.ContextMark



class `contexttools.`**`ContextMark`**(*target_context=None*)

New in version 2.0. Abstract class from which concrete context marks inherit:

```
>>> note = Note("c'4")
```

```
>>> contexttools.ContextMark()(note)
ContextMark()(c'4)
```

Context marks override `__call__` to attach to Abjad components.

Context marks implement `__slots__`.

Read-only properties

`ContextMark.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

ContextMark.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

ContextMark.**storage_format**
Storage format of Abjad object.

Return string.

ContextMark.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Methods

ContextMark.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

ContextMark.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

ContextMark.**__call__**(**args*)

ContextMark.**__copy__**(**args*)

ContextMark.**__deepcopy__**(**args*)

`ContextMark.__delattr__(*args)`

`ContextMark.__eq__(arg)`

`ContextMark.__ge__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`ContextMark.__gt__(expr)`

Abjad objects by default do not implement this method.

Raise exception

`ContextMark.__le__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`ContextMark.__lt__(expr)`

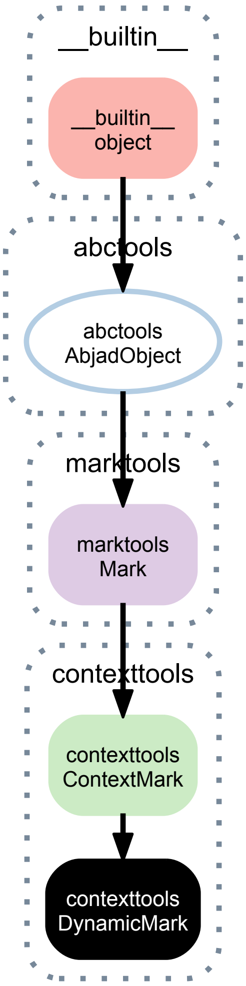
Abjad objects by default do not implement this method.

Raise exception.

`ContextMark.__ne__(arg)`

`ContextMark.__repr__()`

5.1.5 contexttools.DynamicMark



class `contexttools.DynamicMark` (*dynamic_name*, *target_context=None*)

New in version 2.0. Abjad model of a dynamic mark.

Example 1. Initialize from dynamic name:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> contexttools.DynamicMark('f')(staff[0])
DynamicMark('f')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \f
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

Example 2. Initialize from other dynamic mark:

```
>>> dynamic_mark_1 = contexttools.DynamicMark('f')
>>> dynamic_mark_2 = contexttools.DynamicMark(dynamic_mark_1)
```

```
>>> dynamic_mark_1
DynamicMark('f')
```

```
>>> dynamic_mark_2
DynamicMark('f')
```

Dynamic marks target the staff context by default.

Read-only properties

`DynamicMark.effective_context`

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`DynamicMark.lilypond_format`

Read-only LilyPond input format of dynamic mark:

```
>>> dynamic_mark = contexttools.DynamicMark('f')
>>> dynamic_mark.lilypond_format
'\f'
```

Return string.

`DynamicMark.start_component`

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`DynamicMark.`**`storage_format`**

Storage format of Abjad object.

Return string.

`DynamicMark.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`DynamicMark.`**`dynamic_name`**

Get dynamic name:

```
>>> dynamic = contexttools.DynamicMark('f')
>>> dynamic.dynamic_name
'f'
```

Set dynamic name:

```
>>> dynamic.dynamic_name = 'p'
>>> dynamic.dynamic_name
'p'
```

Return string.

Methods

`DynamicMark.`**`attach`**`(`*`start_component`*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`DynamicMark.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

`DynamicMark.`**`__call__`**`(`*`*args`*`)`

`DynamicMark.`**`__copy__`**`(`*`*args`*`)`

`DynamicMark.`**`__deepcopy__`**`(`*`*args`*`)`

`DynamicMark.`**`__delattr__`**`(`**args*`)`

`DynamicMark.`**`__eq__`**`(`*arg*`)`

`DynamicMark.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DynamicMark.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`DynamicMark.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

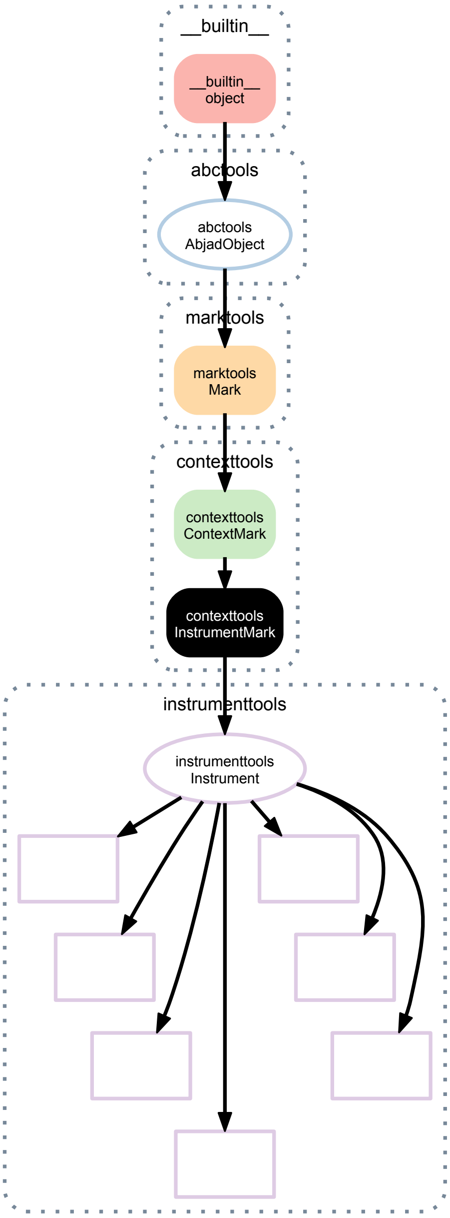
`DynamicMark.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DynamicMark.`**`__ne__`**`(`*arg*`)`

`DynamicMark.`**`__repr__`**`()`

5.1.6 contexttools.InstrumentMark



class `contexttools.InstrumentMark`(*instrument_name*, *short_instrument_name*, *instrument_name_markup=None*, *short_instrument_name_markup=None*, *target_context=None*)

New in version 2.0. Abjad model of an instrument change:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> contexttools.InstrumentMark('Flute', 'Fl.')(staff)
InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Flute }
    \set Staff.shortInstrumentName = \markup { Fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

Instrument marks target staff context by default.

Read-only properties

`InstrumentMark.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`InstrumentMark.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`InstrumentMark.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`InstrumentMark.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`InstrumentMark.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`InstrumentMark.`**`storage_format`**
Storage format of Abjad object.

Return string.

`InstrumentMark.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`InstrumentMark.`**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`InstrumentMark.`**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`InstrumentMark.`**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`InstrumentMark.`**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Methods

`InstrumentMark.`**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

`InstrumentMark.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

`InstrumentMark.`**`__call__`**`(*args)`

`InstrumentMark.`**`__copy__`**`(*args)`

`InstrumentMark.`**`__deepcopy__`**`(*args)`

`InstrumentMark.`**`__delattr__`**`(*args)`

`InstrumentMark.`**`__eq__`**`(arg)`

`InstrumentMark.`**`__ge__`**`(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`InstrumentMark.`**`__gt__`**`(expr)`

Abjad objects by default do not implement this method.

Raise exception

`InstrumentMark.`**`__hash__`**`()`

`InstrumentMark.`**`__le__`**`(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`InstrumentMark.`**`__lt__`**`(expr)`

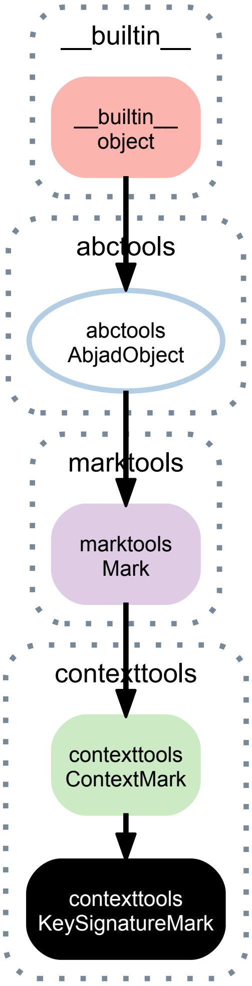
Abjad objects by default do not implement this method.

Raise exception.

`InstrumentMark.`**`__ne__`**`(arg)`

`InstrumentMark.`**`__repr__`**`()`

5.1.7 contexttools.KeySignatureMark



class contexttools.**KeySignatureMark**(*tonic*, *mode*, *target_context=None*)

New in version 2.0. Abjad model of a key signature setting or key signature change:

```
>>> staff = Staff("e'8 fs'8 gs'8 a'8")
```

```
>>> contexttools.KeySignatureMark('e', 'major')(staff)
KeySignatureMark(NamedChromaticPitchClass('e'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key e \major
    e'8
    fs'8
    gs'8
    a'8
}
```

```
>>> show(staff)
```

Key signature marks target staff context by default.

Read-only properties

`KeySignatureMark.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`KeySignatureMark.`**`lilypond_format`**

Read-only LilyPond format of key signature mark:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.lilypond_format
'\\key e \\major'
```

Return string.

`KeySignatureMark.`**`name`**

Read-only name of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.name
'E major'
```

Return string.

`KeySignatureMark.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`KeySignatureMark.`**`storage_format`**

Storage format of Abjad object.

Return string.

`KeySignatureMark.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`KeySignatureMark.`**`mode`**

Get mode of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.mode
Mode('major')
```

Set mode of key signature:

```
>>> key_signature.mode = 'minor'
>>> key_signature.mode
Mode('minor')
```

Return mode.

KeySignatureMark.**tonic**
Get tonic of key signature:

```
>>> key_signature = contexttools.KeySignatureMark('e', 'major')
>>> key_signature.tonic
NamedChromaticPitchClass('e')
```

Set tonic of key signature:

```
>>> key_signature.tonic = 'd'
>>> key_signature.tonic
NamedChromaticPitchClass('d')
```

Return named chromatic pitch.

Methods

KeySignatureMark.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

KeySignatureMark.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

KeySignatureMark.**__call__**(**args*)

KeySignatureMark.**__copy__**(**args*)

KeySignatureMark.**__deepcopy__**(**args*)

KeySignatureMark.**__delattr__**(**args*)

KeySignatureMark.**__eq__**(*arg*)

KeySignatureMark.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

KeySignatureMark.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`KeySignatureMark.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`KeySignatureMark.__lt__(expr)`
Abjad objects by default do not implement this method.

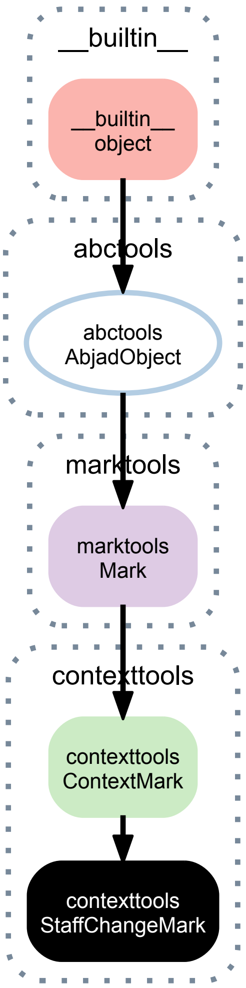
Raise exception.

`KeySignatureMark.__ne__(arg)`

`KeySignatureMark.__repr__()`

`KeySignatureMark.__str__()`

5.1.8 contexttools.StaffChangeMark



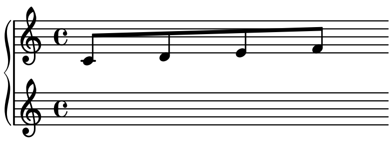
class `contexttools.StaffChangeMark(staff=None, target_context=None)`
New in version 2.0. Abjad model of a staff change:

```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
```

```
>>> f(piano_staff)
\new PianoStaff <<
```

```
        \context Staff = "RHStaff" {
            c'8
            d'8
            e'8
            f'8
        }
        \context Staff = "LHStaff" {
            s2
        }
    >>
```

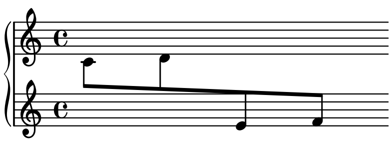
```
>>> show(piano_staff)
```



```
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \context Staff = "RHStaff" {
        c'8
        d'8
        \change Staff = LHStaff
        e'8
        f'8
    }
    \context Staff = "LHStaff" {
        s2
    }
>>
```

```
>>> show(piano_staff)
```



Staff change marks target staff context by default.

Read-only properties

StaffChangeMark.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

StaffChangeMark.**lilypond_format**
Read-only LilyPond format of staff change mark:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff.name = 'RHStaff'
>>> staff_change = contexttools.StaffChangeMark(staff)
>>> staff_change.lilypond_format
'\\change Staff = RHStaff'
```

Return string.

`StaffChangeMark.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`StaffChangeMark.`**`storage_format`**

Storage format of Abjad object.

Return string.

`StaffChangeMark.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`StaffChangeMark.`**`staff`**

Get staff of staff change mark:

```
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> staff_change = contexttools.StaffChangeMark(rh_staff)
>>> staff_change.staff
Staff-"RHStaff"{4}
```

Set staff of staff change mark:

```
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> staff_change.staff = lh_staff
>>> staff_change.staff
Staff-"LHStaff"{1}
```

Return staff.

Methods

`StaffChangeMark.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`StaffChangeMark.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Special methods

StaffChangeMark.**__call__**(*args)

StaffChangeMark.**__copy__**(*args)

StaffChangeMark.**__deepcopy__**(*args)

StaffChangeMark.**__delattr__**(*args)

StaffChangeMark.**__eq__**(arg)

StaffChangeMark.**__ge__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

StaffChangeMark.**__gt__**(expr)
Abjad objects by default do not implement this method.

Raise exception

StaffChangeMark.**__le__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

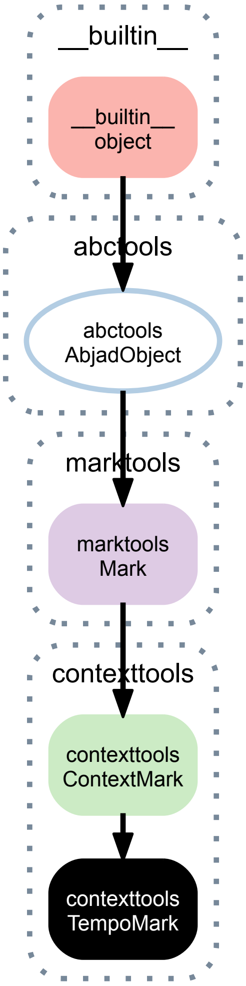
StaffChangeMark.**__lt__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

StaffChangeMark.**__ne__**(arg)

StaffChangeMark.**__repr__**()

5.1.9 contexttools.TempoMark



class `contexttools.TempoMark` (**args*, ***kwargs*)

New in version 2.0. Abjad model of a tempo indication:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff[0])
TempoMark(Duration(1, 8), 52)(c'8)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> show(score)
```

Tempo marks target **score** context by default.

Initialization allows many different types of input argument structure.

Read-only properties

TempoMark.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

TempoMark.**is_imprecise**

True if tempo mark is entirely textual, or if tempo mark's units_per_minute is a range:

```
>>> contexttools.TempoMark(Duration(1, 4), 60).is_imprecise
False
>>> contexttools.TempoMark('Langsam', 4, 60).is_imprecise
False
>>> contexttools.TempoMark('Langsam').is_imprecise
True
>>> contexttools.TempoMark('Langsam', 4, (35, 50)).is_imprecise
True
>>> contexttools.TempoMark(Duration(1, 4), (35, 50)).is_imprecise
True
```

Return boolean.

TempoMark.**lilypond_format**

Read-only LilyPond format of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.lilypond_format
'\\tempo 8=52'
```

```
>>> tempo.textual_indication = 'Gingerly'
>>> tempo.lilypond_format
'\\tempo Gingerly 8=52'
```

```
>>> tempo.units_per_minute = (52, 56)
>>> tempo.lilypond_format
'\\tempo Gingerly 8=52~56'
```

Return string.

TempoMark.**quarters_per_minute**

Read-only quarters per minute of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.quarters_per_minute
Fraction(104, 1)
```

Return fraction.

Or tuple if tempo mark *units_per_minute* is a range.

Or none if tempo mark is imprecise.

TempoMark.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`TempoMark.`**storage_format**

Storage format of Abjad object.

Return string.

`TempoMark.`**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Read/write properties

`TempoMark.`**duration**

Get duration of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.duration
Duration(1, 8)
```

Set duration of tempo mark:

```
>>> tempo.duration = Duration(1, 4)
>>> tempo.duration
Duration(1, 4)
```

Return duration, or None if tempo mark is imprecise.

`TempoMark.`**textual_indication**

Get textual indication of tempo mark:

```
>>> tempo = contexttools.TempoMark('Langsam', Duration(1, 8), 52)
>>> tempo.textual_indication
'Langsam'
```

Return string or None.

`TempoMark.`**units_per_minute**

Get units per minute of tempo mark:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo.units_per_minute
52
```

Set units per minute of tempo mark:

```
>>> tempo.units_per_minute = 56
>>> tempo.units_per_minute
56
```

Return number.

Methods

`TempoMark.`**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

`TempoMark.`**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`TempoMark.`**`is_tempo_mark_token`**`(`*expr*`)`
True when *expr* has the form of a tempo mark initializer:

```
>>> tempo_mark = contexttools.TempoMark(Duration(1, 4), 72)
>>> tempo_mark.is_tempo_mark_token((Duration(1, 4), 84))
True
```

Otherwise false:

```
>>> tempo_mark.is_tempo_mark_token(84)
False
```

Return boolean.

Special methods

`TempoMark.`**`__add__`**`(`*expr*`)`

`TempoMark.`**`__call__`**`(`**args*`)`

`TempoMark.`**`__copy__`**`(`**args*`)`

`TempoMark.`**`__deepcopy__`**`(`**args*`)`

`TempoMark.`**`__delattr__`**`(`**args*`)`

`TempoMark.`**`__div__`**`(`*expr*`)`

`TempoMark.`**`__eq__`**`(`*expr*`)`

`TempoMark.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TempoMark.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`TempoMark.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TempoMark.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

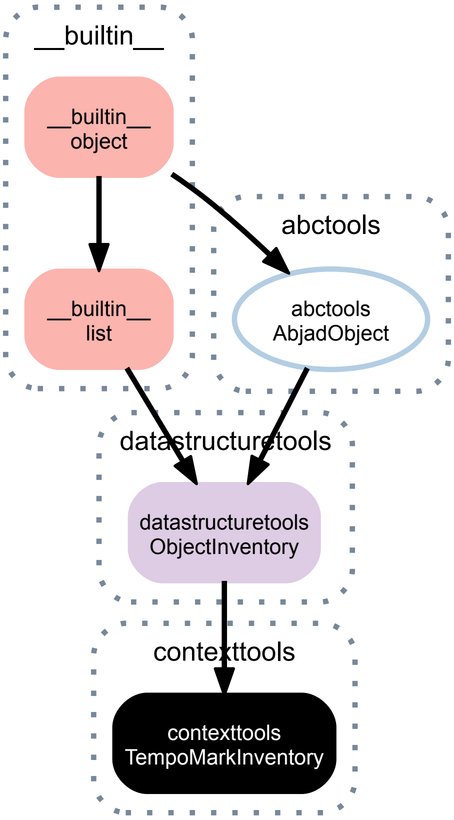
`TempoMark.`**`__mul__`**`(`*multiplier*`)`

`TempoMark.`**`__ne__`**`(`*arg*`)`

`TempoMark.`**`__repr__`**`()`

`TempoMark.`**`__sub__`**`(`*expr*`)`

5.1.10 contexttools.TempoMarkInventory



class contexttools.**TempoMarkInventory**(*tokens=None*, *name=None*)

New in version 2.7. Abjad model of an ordered list of tempo marks:

```
>>> inventory = contexttools.TempoMarkInventory([
...     ('Andante', Duration(1, 8), 72),
...     ('Allegro', Duration(1, 8), 84)])
```

```
>>> for tempo_mark in inventory:
...     tempo_mark
...
TempoMark('Andante', Duration(1, 8), 72)
TempoMark('Allegro', Duration(1, 8), 84)
```

Tempo mark inventories implement list interface and are mutable.

Read-only properties

TempoMarkInventory.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

TempoMarkInventory.**name**

Read / write name of inventory.

Methods

`TempoMarkInventory.append(token)`
Change *token* to item and append.

`TempoMarkInventory.count(value)` → integer – return number of occurrences of value

`TempoMarkInventory.extend(tokens)`
Change *tokens* to items and extend.

`TempoMarkInventory.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`TempoMarkInventory.insert()`
L.insert(index, object) – insert object before index

`TempoMarkInventory.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`TempoMarkInventory.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`TempoMarkInventory.reverse()`
L.reverse() – reverse *IN PLACE*

`TempoMarkInventory.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`TempoMarkInventory.__add__()`
x.__add__(y) <==> x+y

`TempoMarkInventory.__contains__(token)`

`TempoMarkInventory.__delitem__()`
x.__delitem__(y) <==> del x[y]

`TempoMarkInventory.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`TempoMarkInventory.__eq__()`
x.__eq__(y) <==> x==y

`TempoMarkInventory.__ge__()`
x.__ge__(y) <==> x>=y

`TempoMarkInventory.__getitem__()`
x.__getitem__(y) <==> x[y]

`TempoMarkInventory.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`TempoMarkInventory.__gt__()`
x.__gt__(y) <==> x>y

`TempoMarkInventory.__iadd__()`
x.__iadd__(y) <==> x+=y

`TempoMarkInventory.__imul__()`
x.__imul__(y) <==> x*=y

`TempoMarkInventory.__iter__()` <==> *iter(x)*

`TempoMarkInventory.__le__()`
x.__le__(y) <==> x<=y

`TempoMarkInventory.__len__()` <==> *len(x)*

`TempoMarkInventory.__lt__()`
x.__lt__(y) <==> x<y

`TempoMarkInventory.__mul__()`
x.__mul__(n) <==> x*n

`TempoMarkInventory.__ne__()`
x.__ne__(y) <==> x!=y

`TempoMarkInventory.__repr__()`

`TempoMarkInventory.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

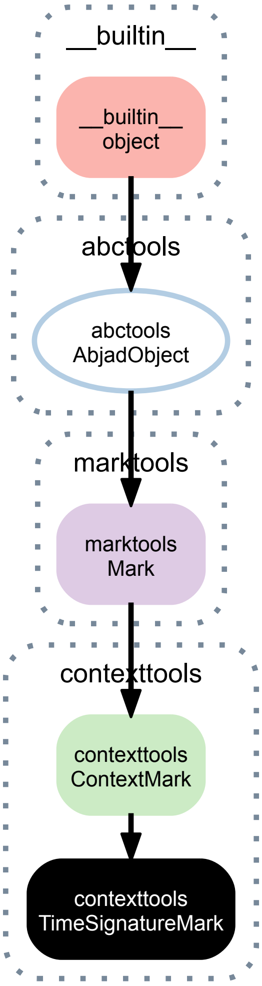
`TempoMarkInventory.__rmul__()`
x.__rmul__(n) <==> n*x

`TempoMarkInventory.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`TempoMarkInventory.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

5.1.11 contexttools.TimeSignatureMark



class `contexttools.`**`TimeSignatureMark`**(**args*, ***kwargs*)

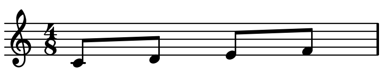
New in version 2.0. Abjad model of a time signature:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> contexttools.TimeSignatureMark((4, 8))(staff[0])
TimeSignatureMark((4, 8))(c'8)
```

```
>>> f(staff)
\new Staff {
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



Abjad time signature marks target **staff context** by default.

Initialize time signature marks to **score context** like this:

```
>>> contexttools.TimeSignatureMark((4, 8), target_context=Score)
TimeSignatureMark((4, 8), target_context=Score)
```

Note: add more initializer examples.

Note: add example of *suppress* keyword.

Note: turn *suppress* into managed attribute.

Return time signature object.

Read-only properties

TimeSignatureMark.**duration**

Time signature mark duration:

```
>>> contexttools.TimeSignatureMark((3, 8)).duration
Duration(3, 8)
```

Return duration.

TimeSignatureMark.**effective_context**

Time signature mark effective context.

Return none when time signature mark is not yet attached:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
```

```
>>> time_signature.effective_context is None
True
```

Return context when time signature mark is attached:

```
>>> staff = Staff()
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.effective_context
Staff{}
```

Return context or none.

TimeSignatureMark.**has_non_power_of_two_denominator**

True when time signature mark has non-power-of-two denominator:

```
>>> contexttools.TimeSignatureMark((7, 12)).has_non_power_of_two_denominator
True
```

Otherwise false:

```
>>> contexttools.TimeSignatureMark((3, 8)).has_non_power_of_two_denominator
False
```

Return boolean.

TimeSignatureMark.**implied_prolation**

New in version 2.11. Time signature mark implied prolation.

Example 1. Implied prolation of time signature with power-of-two denominator:

```
>>> contexttools.TimeSignatureMark((3, 8)).implied_prolation
Multiplier(1, 1)
```

Example 2. Implied prolation of time signature with non-power-of-two denominator:

```
>>> contexttools.TimeSignatureMark((7, 12)).implied_prolation
Multiplier(2, 3)
```

Return multiplier.

TimeSignatureMark.**lilypond_format**
Time signature mark LilyPond format:

```
>>> contexttools.TimeSignatureMark((3, 8)).lilypond_format
'\\time 3/8'
```

Return string.

TimeSignatureMark.**pair**
New in version 2.8. Time signature numerator / denominator pair:

```
>>> contexttools.TimeSignatureMark((3, 8)).pair
(3, 8)
```

Return pair.

TimeSignatureMark.**start_component**
Time signature mark start component.

Return none when time signature mark is not yet attached:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.start_component is None
True
```

Return component when time signature mark is attached:

```
>>> staff = Staff()
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.start_component
Staff{}
```

Return component or none.

TimeSignatureMark.**storage_format**
Time signature mark storage format:

```
>>> print contexttools.TimeSignatureMark((3, 8)).storage_format
contexttools.TimeSignatureMark(
    (3, 8)
    )
```

Return string.

TimeSignatureMark.**target_context**
Time signature mark target context:

```
>>> contexttools.TimeSignatureMark((3, 8)).target_context
<class 'abjad.tools.stafftools.Staff.Staff.Staff'>
```

Time signature marks target the staff context by default.

This can be changed at initialization.

Return class.

Read/write properties

TimeSignatureMark.**denominator**
Get denominator of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
```

```
>>> time_signature.denominator
8
```

Set denominator of time signature mark:

```
>>> time_signature.denominator = 16
```

```
>>> time_signature.denominator
16
```

Return integer.

`TimeSignatureMark.`**`numerator`**

Get numerator of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> time_signature.numerator
3
```

Set numerator of time signature mark:

```
>>> time_signature.numerator = 4
>>> time_signature.numerator
4
```

Set integer.

`TimeSignatureMark.`**`partial`**

Get partial measure pick-up of time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark(
...     (3, 8), partial=Duration(1, 8))
>>> time_signature.partial
Duration(1, 8)
```

Set partial measure pick-up of time signature mark:

```
>>> time_signature.partial = Duration(1, 4)
>>> time_signature.partial
Duration(1, 4)
```

Set duration or none.

Methods

`TimeSignatureMark.`**`attach`**(*start_component*)

Attach time signature mark to *start_component*:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> staff = Staff()
```

```
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

Return time signature mark.

`TimeSignatureMark.`**`detach`**()

Detach time signature mark:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 8))
>>> staff = Staff()
```

```
>>> time_signature.attach(staff)
TimeSignatureMark((3, 8))(Staff{})
```

```
>>> time_signature.detach()
TimeSignatureMark((3, 8))
```

Return time signature mark.

TimeSignatureMark.**with_power_of_two_denominator**(*contents_multiplier=Multiplier(1, 1)*)

Create new time signature equivalent to current time signature with power-of-two denominator.

```
>>> time_signature = contexttools.TimeSignatureMark((3, 12))
```

```
>>> time_signature.with_power_of_two_denominator()
TimeSignatureMark((2, 8))
```

Return new time signature mark.

Special methods

TimeSignatureMark.**__call__**(**args*)

TimeSignatureMark.**__copy__**(**args*)

TimeSignatureMark.**__deepcopy__**(**args*)

TimeSignatureMark.**__delattr__**(**args*)

TimeSignatureMark.**__eq__**(*arg*)

TimeSignatureMark.**__ge__**(*arg*)

TimeSignatureMark.**__gt__**(*arg*)

TimeSignatureMark.**__le__**(*arg*)

TimeSignatureMark.**__lt__**(*arg*)

TimeSignatureMark.**__ne__**(*arg*)

TimeSignatureMark.**__nonzero__**()

TimeSignatureMark.**__repr__**()

TimeSignatureMark.**__str__**()

5.2 Functions

5.2.1 contexttools.all_are_contexts

contexttools.**all_are_contexts**(*expr*)

New in version 2.10. True when *expr* is a sequence of Abjad contexts:

```
>>> contexts = 3 * Voice("c'8 d'8 e'8")
```

```
>>> contexttools.all_are_contexts(contexts)
True
```

True when *expr* is an empty sequence:

```
>>> contexttools.all_are_contexts([])
True
```

Otherwise false:

```
>>> contexttools.all_are_contexts('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

5.2.2 contexttools.detach_clef_marks_attached_to_component

`contexttools.`**`detach_clef_marks_attached_to_component`**`(`*component*`)`

New in version 2.3. Detach clef marks attached to *component*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> clef_mark = contexttools.ClefMark('treble')
>>> clef_mark.attach(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> contexttools.detach_clef_marks_attached_to_component(staff)
(ClefMark('treble'),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more clef marks.

5.2.3 contexttools.detach_context_marks_attached_to_component

`contexttools.`**`detach_context_marks_attached_to_component`**`(`*component*, *klasses=None*`)`

New in version 2.0. Detach context marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> clef_mark = contexttools.ClefMark('treble')(staff)
>>> dynamic_mark = contexttools.DynamicMark('p')(staff[0])
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8 \p
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.detach_context_marks_attached_to_component(staff[0])
(DynamicMark('p'),)
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
    e'8
    f'8
}
```

Return tuple of zero or context marks.

5.2.4 contexttools.detach_dynamic_marks_attached_to_component

contexttools.**detach_dynamic_marks_attached_to_component**(*component*)
New in version 2.3. Detach dynamic marks attached to *component*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> dynamic_mark = contexttools.DynamicMark('p')
>>> dynamic_mark.attach(staff[0])
DynamicMark('p')(c'4)
```

```
>>> f(staff)
\new Staff {
    c'4 \p
    d'4
    e'4
    f'4
}
```

```
>>> contexttools.detach_dynamic_marks_attached_to_component(staff[0])
(DynamicMark('p'),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more dynamic marks.

5.2.5 contexttools.detach_instrument_marks_attached_to_component

contexttools.**detach_instrument_marks_attached_to_component**(*component*)
New in version 2.1. Detach instrument marks attached to *component*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> instrument_mark = contexttools.InstrumentMark('Violin ', 'Vn. ')
>>> instrument_mark.attach(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Violin  }
    \set Staff.shortInstrumentName = \markup { Vn.  }
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> contexttools.detach_instrument_marks_attached_to_component(staff)
(InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. '),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more instrument marks.

5.2.6 contexttools.detach_key_signature_marks_attached_to_component

contexttools.**detach_key_signature_marks_attached_to_component**(*component*)
New in version 2.3. Detach key signature marks attached to *component*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> key_signature_mark = contexttools.KeySignatureMark('c', 'major')
>>> key_signature_mark.attach(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key c \major
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> contexttools.detach_key_signature_marks_attached_to_component(staff)
(KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major')),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more key signature marks.

5.2.7 contexttools.detach_staff_change_marks_attached_to_component

contexttools.**detach_staff_change_marks_attached_to_component**(*component*)
New in version 2.3. Detach staff change marks attached to *component*:

```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \context Staff = "RHStaff" {
        c'8
        d'8
        \change Staff = LHStaff
        e'8
        f'8
    }
    \context Staff = "LHStaff" {
        s2
    }
>>
```

```
>>> contexttools.detach_staff_change_marks_attached_to_component(rh_staff[2])
(StaffChangeMark(Staff-"LHStaff"{1}),)
```

Return tuple of zero or more staff change marks.

5.2.8 contexttools.detach_tempo_marks_attached_to_component

`contexttools.detach_tempo_marks_attached_to_component(component)`

New in version 2.3. Detach tempo marks attached to *component*:

```
>>> score = Score([])
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score.append(staff)
```

```
>>> tempo_mark = contexttools.TempoMark(Duration(1, 8), 52)
>>> tempo_mark.attach(staff)
TempoMark(Duration(1, 8), 52)(Staff{4})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'4
        d'4
        e'4
        f'4
    }
>>
```

```
>>> contexttools.detach_tempo_marks_attached_to_component(staff)
(TempoMark(Duration(1, 8), 52),)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
>>
```

Return tuple of zero or more tempo marks.

5.2.9 contexttools.detach_time_signature_marks_attached_to_component

`contexttools.detach_time_signature_marks_attached_to_component(component)`

New in version 2.0. Detach time signature marks attached to *component*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> contexttools.TimeSignatureMark((4, 4))(staff[0])
TimeSignatureMark((4, 4))(c'4)
```

```
>>> f(staff)
\new Staff {
    \time 4/4
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> contexttools.detach_time_signature_marks_attached_to_component(staff[0])
(TimeSignatureMark((4, 4)),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more time signature marks.

5.2.10 contexttools.get_clef_mark_attached_to_component

contexttools.**get_clef_mark_attached_to_component**(*component*)
New in version 2.3. Get clef mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_clef_mark_attached_to_component(staff)
ClefMark('treble')(Staff{4})
```

Return clef mark.

Raise missing mark error when no clef mark attached to *component*.

5.2.11 contexttools.get_clef_marks_attached_to_component

contexttools.**get_clef_marks_attached_to_component**(*component*)
New in version 2.3. Get clef marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_clef_marks_attached_to_component(staff)
(ClefMark('treble')(Staff{4}),)
```

Return tuple of zero or more clef marks.

5.2.12 contexttools.get_context_mark_attached_to_component

contexttools.**get_context_mark_attached_to_component**(*component*, *klasses=None*)
New in version 2.3. Get context mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
```

```
        e'8
        f'8
}
```

```
>>> contexttools.get_context_mark_attached_to_component(staff)
ClefMark('treble')(Staff{4})
```

Return context mark.

Raise missing mark error when no context mark attaches to *component*.

5.2.13 contexttools.get_context_marks_attached_to_any_improper_parent_of_component

contexttools.**get_context_marks_attached_to_any_improper_parent_of_component**(*component*)
New in version 2.0. Get all context marks attached to any improper parent of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
>>> contexttools.DynamicMark('f')(staff[0])
DynamicMark('f')(c'8)
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8 \f
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_context_marks_attached_to_any_improper_parent_of_component(staff[0])
(DynamicMark('f')(c'8), ClefMark('treble')(Staff{4}))
```

Return tuple.

5.2.14 contexttools.get_context_marks_attached_to_component

contexttools.**get_context_marks_attached_to_component**(*component*, *klasses=None*)
New in version 2.0. Get context marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
>>> contexttools.DynamicMark('p')(staff[0])
DynamicMark('p')(c'8)
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8 \p
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_context_marks_attached_to_component(staff[0])
(DynamicMark('p')(c'8),)
```

Return tuple of zero or more context marks.

5.2.15 contexttools.get_dynamic_mark_attached_to_component

contexttools.**get_dynamic_mark_attached_to_component**(*component*)
New in version 2.3. Get dynamic mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.DynamicMark('p')(staff[0])
DynamicMark('p')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \p
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_dynamic_mark_attached_to_component(staff[0])
DynamicMark('p')(c'8)
```

Return dynamic mark.

Raise missing mark error when no dynamic mark attaches to *component*.

Raise extra mark error when more than one dynamic mark attaches to *component*.

5.2.16 contexttools.get_dynamic_marks_attached_to_component

contexttools.**get_dynamic_marks_attached_to_component**(*component*)
New in version 2.0. Get dynamic marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.DynamicMark('p')(staff[0])
DynamicMark('p')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \p
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_dynamic_marks_attached_to_component(staff[0])
(DynamicMark('p')(c'8),)
```

Return tuple of zero or more dynamic marks.

5.2.17 contexttools.get_effective_clef

contexttools.**get_effective_clef**(*component*)
New in version 2.0. Get effective clef of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> for note in staff:
...     print note, contexttools.get_effective_clef(note)
...
c'8 ClefMark('treble')(Staff{4})
d'8 ClefMark('treble')(Staff{4})
e'8 ClefMark('treble')(Staff{4})
f'8 ClefMark('treble')(Staff{4})
```

Return clef mark or none.

5.2.18 contexttools.get_effective_context_mark

`contexttools.get_effective_context_mark(component, klass)`

New in version 2.0. Get effective context mark of *klass* from *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((4, 8))(staff)
TimeSignatureMark((4, 8))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_effective_context_mark(staff[0], contexttools.TimeSignatureMark)
TimeSignatureMark((4, 8))(Staff{4})
```

Return context mark or none.

5.2.19 contexttools.get_effective_dynamic

`contexttools.get_effective_dynamic(component)`

New in version 2.0. Get effective dynamic of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.DynamicMark('f')(staff[0])
DynamicMark('f')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \f
    d'8
    e'8
    f'8
}
```

```
>>> for note in staff:
...     print note, contexttools.get_effective_dynamic(note)
...
c'8 DynamicMark('f')(c'8)
d'8 DynamicMark('f')(c'8)
e'8 DynamicMark('f')(c'8)
f'8 DynamicMark('f')(c'8)
```

Return dynamic mark or none.

5.2.20 contexttools.get_effective_instrument

`contexttools.get_effective_instrument(component)`

New in version 2.0. Get effective instrument of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.InstrumentMark('Flute', 'Fl.')(staff)
InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Flute }
    \set Staff.shortInstrumentName = \markup { Fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> for note in staff:
...     print note, contexttools.get_effective_instrument(note)
...
c'8 InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
d'8 InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
e'8 InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
f'8 InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
```

Return instrument mark or none.

5.2.21 contexttools.get_effective_key_signature

contexttools.**get_effective_key_signature**(*component*)

New in version 2.0. Get effective key signature of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.KeySignatureMark('c', 'major')(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key c \major
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> for note in staff:
...     note, contexttools.get_effective_key_signature(note)
...
(Note("c'8"), KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4}))
(Note("d'8"), KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4}))
(Note("e'8"), KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4}))
(Note("f'8"), KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4}))
```

Return key signature mark or none.

5.2.22 contexttools.get_effective_staff

contexttools.**get_effective_staff**(*component*)

New in version 2.0. Get effective staff of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff.name = 'First Staff'
```

```
>>> f(staff)
\context Staff = "First Staff" {
    c'8
    d'8
    e'8
```

```
        f'8
    }
```

```
>>> for note in staff:
...     print note, contexttools.get_effective_staff(note)
...
c'8 Staff-"First Staff"{4}
d'8 Staff-"First Staff"{4}
e'8 Staff-"First Staff"{4}
f'8 Staff-"First Staff"{4}
```

Return staff or none.

5.2.23 contexttools.get_effective_tempo

contexttools.**get_effective_tempo**(*component*)

New in version 2.0. Get effective tempo of *component*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff[0])
TempoMark(Duration(1, 8), 52)(c'8)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> for note in staff:
...     print note, contexttools.get_effective_tempo(note)
...
c'8 TempoMark(Duration(1, 8), 52)(c'8)
d'8 TempoMark(Duration(1, 8), 52)(c'8)
e'8 TempoMark(Duration(1, 8), 52)(c'8)
f'8 TempoMark(Duration(1, 8), 52)(c'8)
```

Return tempo mark or none.

5.2.24 contexttools.get_effective_time_signature

contexttools.**get_effective_time_signature**(*component*)

New in version 2.0. Get effective time signature of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((4, 8))(staff)
TimeSignatureMark((4, 8))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> for note in staff:
...     note, contexttools.get_effective_time_signature(note)
...
```

```
(Note("c'8"), TimeSignatureMark((4, 8))(Staff{4}))
(Note("d'8"), TimeSignatureMark((4, 8))(Staff{4}))
(Note("e'8"), TimeSignatureMark((4, 8))(Staff{4}))
(Note("f'8"), TimeSignatureMark((4, 8))(Staff{4}))
```

Return time signature mark or none.

5.2.25 contexttools.get_instrument_mark_attached_to_component

contexttools.**get_instrument_mark_attached_to_component**(*component*)
New in version 2.1. Get instrument mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> violin = contexttools.InstrumentMark('Violin ', 'Vn. ')
>>> violin.attach(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Violin  }
    \set Staff.shortInstrumentName = \markup { Vn.  }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_instrument_mark_attached_to_component(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

Return instrument mark.

Raise missing mark error when no instrument mark attaches to *component*.

5.2.26 contexttools.get_instrument_marks_attached_to_component

contexttools.**get_instrument_marks_attached_to_component**(*component*)
New in version 2.3. Get instrument marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.InstrumentMark('Flute', 'Fl.')(staff)
InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Flute }
    \set Staff.shortInstrumentName = \markup { Fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_instrument_marks_attached_to_component(staff)
(InstrumentMark(instrument_name='Flute', short_instrument_name='Fl.')(Staff{4}),)
```

Return tuple of zero or more instrument marks.

5.2.27 contexttools.get_key_signature_mark_attached_to_component

contexttools.**get_key_signature_mark_attached_to_component**(*component*)
New in version 2.3. Get key signature mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.KeySignatureMark('c', 'major')(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key c \major
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_key_signature_mark_attached_to_component(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

Return key signature mark.

Raise missing mark error when no key signature mark attaches to component.

5.2.28 contexttools.get_key_signature_marks_attached_to_component

contexttools.**get_key_signature_marks_attached_to_component**(*component*)
New in version 2.3. Get key signature marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.KeySignatureMark('c', 'major')(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key c \major
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_key_signature_marks_attached_to_component(staff)
(KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4}),)
```

Return tuple of zero or more key signature marks.

5.2.29 contexttools.get_staff_change_mark_attached_to_component

contexttools.**get_staff_change_mark_attached_to_component**(*component*)
New in version 2.3. Get staff change mark attached to *component*:

```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \context Staff = "RHStaff" {
        c'8
        d'8
        \change Staff = LHStaff
        e'8
        f'8
    }
```

```
        \context Staff = "LHStaff" {
            s2
        }
    >>
```

```
>>> contexttools.get_staff_change_mark_attached_to_component(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

Return staff change mark.

Raise missing mark error when no staff change mark attaches to *component*.

5.2.30 contexttools.get_staff_change_marks_attached_to_component

contexttools.**get_staff_change_marks_attached_to_component**(*component*)

New in version 2.3. Get staff change marks attached to *component*:

```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \context Staff = "RHStaff" {
        c'8
        d'8
        \change Staff = LHStaff
        e'8
        f'8
    }
    \context Staff = "LHStaff" {
        s2
    }
>>
```

```
>>> contexttools.get_staff_change_marks_attached_to_component(rh_staff[2])
(StaffChangeMark(Staff-"LHStaff"{1})(e'8),)
```

Return tuple of zero or more staff change marks.

5.2.31 contexttools.get_tempo_mark_attached_to_component

contexttools.**get_tempo_mark_attached_to_component**(*component*)

New in version 2.3. Get tempo mark attached to *component*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff)
TempoMark(Duration(1, 8), 52)(Staff{4})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> contexttools.get_tempo_mark_attached_to_component(staff)
TempoMark(Duration(1, 8), 52)(Staff{4})
```

Return tempo mark.

Raise missing mark error when no tempo mark attaches to *component*.

5.2.32 contexttools.get_tempo_marks_attached_to_component

contexttools.**get_tempo_marks_attached_to_component**(*component*)
New in version 2.3. Get tempo marks attached to *component*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff)
TempoMark(Duration(1, 8), 52)(Staff{4})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> contexttools.get_tempo_marks_attached_to_component(staff)
(TempoMark(Duration(1, 8), 52)(Staff{4}),)
```

Return tuple of zero or more tempo marks.

5.2.33 contexttools.get_time_signature_mark_attached_to_component

contexttools.**get_time_signature_mark_attached_to_component**(*component*)
New in version 2.0. Get time signature mark attached to *component*:

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
```

```
>>> f(measure)
{
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_time_signature_mark_attached_to_component(measure)
TimeSignatureMark((4, 8))(|4/8 c'8 d'8 e'8 f'8|)
```

Return time signature mark.

Raise missing mark error when no time signature mark attaches to *component*.

5.2.34 contexttools.get_time_signature_marks_attached_to_component

contexttools.**get_time_signature_marks_attached_to_component**(*component*)
New in version 2.3. Get time signature marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((2, 4))(staff)
TimeSignatureMark((2, 4))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.get_time_signature_marks_attached_to_component(staff)
(TimeSignatureMark((2, 4))(Staff{4}),)
```

Return tuple of zero or more time_signature marks.

5.2.35 contexttools.is_component_with_clef_mark_attached

contexttools.**is_component_with_clef_mark_attached**(*expr*)
New in version 2.3. True when *expr* is a component with clef mark attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "treble"
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.is_component_with_clef_mark_attached(staff)
True
```

False otherwise:

```
>>> contexttools.is_component_with_clef_mark_attached(staff[0])
False
```

Return boolean.

5.2.36 contexttools.is_component_with_context_mark_attached

contexttools.**is_component_with_context_mark_attached**(*expr*, *klasses=None*)
New in version 2.0. True when *expr* is a component with context mark of *klasses* attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((4, 8))(staff[0])
TimeSignatureMark((4, 8))(c'8)
>>> f(staff)
\new Staff {
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
>>> contexttools.is_component_with_context_mark_attached(staff[0])
True
```

Otherwise false:

```
>>> contexttools.is_component_with_context_mark_attached(staff)
False
```

Return boolean.

5.2.37 contexttools.is_component_with_dynamic_mark_attached

contexttools.**is_component_with_dynamic_mark_attached**(*expr*)

New in version 2.3. True when *expr* is a component and has a dynamic mark attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.DynamicMark('p')(staff[0])
DynamicMark('p')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \p
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.is_component_with_dynamic_mark_attached(staff[0])
True
```

Otherwise false:

```
>>> contexttools.is_component_with_dynamic_mark_attached(staff)
False
```

Return boolean.

5.2.38 contexttools.is_component_with_instrument_mark_attached

contexttools.**is_component_with_instrument_mark_attached**(*expr*)

New in version 2.3. True when *expr* is a component with instrument mark attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> violin = contexttools.InstrumentMark('Violin ', 'Vn. ')
>>> violin.attach(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Violin  }
    \set Staff.shortInstrumentName = \markup { Vn.  }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.is_component_with_instrument_mark_attached(staff)
True
```

Otherwise false:

```
>>> contexttools.is_component_with_instrument_mark_attached(staff[0])
False
```

Return boolean.

5.2.39 contexttools.is_component_with_key_signature_mark_attached

`contexttools.is_component_with_key_signature_mark_attached(expr)`

New in version 2.3. True when *expr* is a component with key signature mark attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.KeySignatureMark('c', 'major')(staff)
KeySignatureMark(NamedChromaticPitchClass('c'), Mode('major'))(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \key c \major
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.is_component_with_key_signature_mark_attached(staff)
True
```

Otherwise false:

```
>>> contexttools.is_component_with_key_signature_mark_attached(staff[0])
False
```

Return boolean.

5.2.40 contexttools.is_component_with_staff_change_mark_attached

`contexttools.is_component_with_staff_change_mark_attached(expr)`

New in version 2.3. True when *expr* is a component with staff change mark attached:

```
>>> piano_staff = scoretools.PianoStaff([])
>>> rh_staff = Staff("c'8 d'8 e'8 f'8")
>>> rh_staff.name = 'RHStaff'
>>> lh_staff = Staff("s2")
>>> lh_staff.name = 'LHStaff'
>>> piano_staff.extend([rh_staff, lh_staff])
>>> contexttools.StaffChangeMark(lh_staff)(rh_staff[2])
StaffChangeMark(Staff-"LHStaff"{1})(e'8)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \context Staff = "RHStaff" {
        c'8
        d'8
        \change Staff = LHStaff
        e'8
        f'8
    }
    \context Staff = "LHStaff" {
        s2
    }
>>
```

```
>>> contexttools.is_component_with_staff_change_mark_attached(rh_staff[2])
True
```

Otherwise false:

```
>>> contexttools.is_component_with_staff_change_mark_attached(rh_staff)
False
```

Return boolean.

5.2.41 contexttools.is_component_with_tempo_mark_attached

contexttools.**is_component_with_tempo_mark_attached**(*expr*)

New in version 2.3. True when *expr* is a component with tempo mark attached:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> contexttools.TempoMark(Duration(1, 8), 52)(staff)
TempoMark(Duration(1, 8), 52)(Staff{4})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \tempo 8=52
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> contexttools.is_component_with_tempo_mark_attached(staff)
True
```

Otherwise false:

```
>>> contexttools.is_component_with_tempo_mark_attached(staff[0])
False
```

Return boolean.

5.2.42 contexttools.is_component_with_time_signature_mark_attached

contexttools.**is_component_with_time_signature_mark_attached**(*expr*)

New in version 2.0. True when *expr* is a component with time signature mark attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((4, 8))(staff[0])
TimeSignatureMark((4, 8))(c'8)
```

```
>>> f(staff)
\new Staff {
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> contexttools.is_component_with_time_signature_mark_attached(staff[0])
True
```

Otherwise false:

```
>>> contexttools.is_component_with_time_signature_mark_attached(staff)
False
```

Return boolean.

5.2.43 contexttools.list_clef_names

contexttools.**list_clef_names**()

New in version 2.8. List clef names:

```
>>> contexttools.list_clef_names()
['alto', 'baritone', 'bass', 'mezzosoprano', 'percussion', 'soprano', 'treble']
```

Return list of strings.

5.2.44 contexttools.set_accidental_style_on_sequential_contexts_in_expr

contexttools.**set_accidental_style_on_sequential_contexts_in_expr**(*expr*, *accidental_style*)

New in version 2.0. Set *accidental_style* for sequential semantic contexts in *expr*:

```
>>> score = Score(Staff("c'8 d'8") * 2)
>>> contexttools.set_accidental_style_on_sequential_contexts_in_expr(score, 'forget')
```

```
>>> f(score)
\new Score <<
    \new Staff {
        #(set-accidental-style 'forget)
        c'8
        d'8
    }
    \new Staff {
        #(set-accidental-style 'forget)
        c'8
        d'8
    }
>>
```

Skip nonsemantic contexts.

Function looks like a hack but isn't. LilyPond uses the dedicated command shown here to set accidental style. This means that it is not possible to set accidental style on a top-level context like score with a single override.

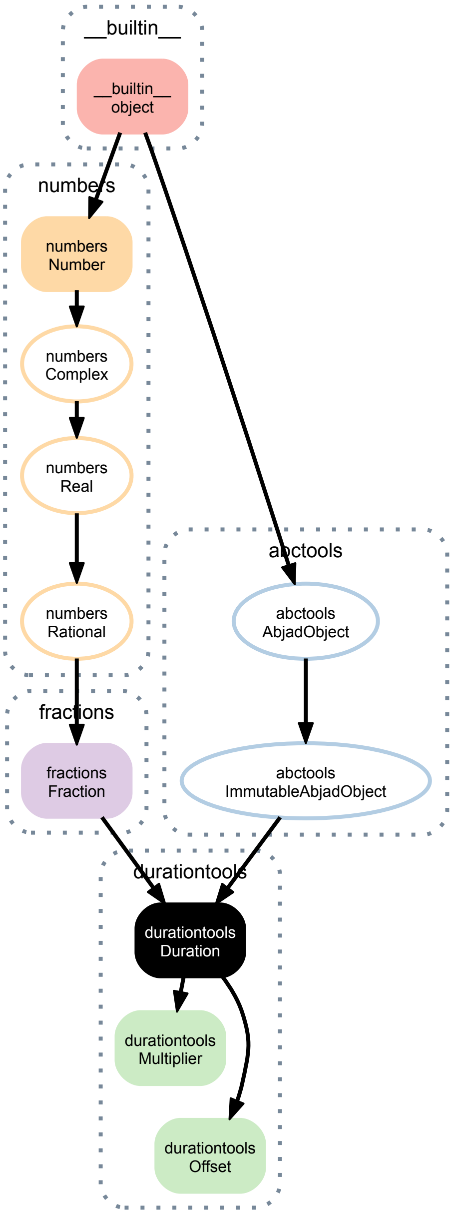
CHAPTER

SIX

DURATIONTOOLS

6.1 Concrete Classes

6.1.1 durationtools.Duration



class durationtools.**Duration**(**args*, ***kwargs*)

New in version 2.0. Abjad model of musical duration.

Initialize from integer numerator:

```
>>> Duration(3)
Duration(3, 1)
```

Initialize from integer numerator and denominator:

```
>>> Duration(3, 16)
Duration(3, 16)
```

Initialize from integer-equivalent numeric numerator:

```
>>> Duration(3.0)
Duration(3, 1)
```

Initialize from integer-equivalent numeric numerator and denominator:

```
>>> Duration(3.0, 16)
Duration(3, 16)
```

Initialize from integer-equivalent singleton:

```
>>> Duration((3,))
Duration(3, 1)
```

Initialize from integer-equivalent pair:

```
>>> Duration((3, 16))
Duration(3, 16)
```

Initialize from other duration:

```
>>> Duration(Duration(3, 16))
Duration(3, 16)
```

Intialize from fraction:

```
>>> Duration(Fraction(3, 16))
Duration(3, 16)
```

Initialize from solidus string:

```
>>> Duration('3/16')
Duration(3, 16)
```

Initialize from nonreduced fraction:

```
>>> Duration(mathtools.NonreducedFraction(3, 16))
Duration(3, 16)
```

Durations inherit from built-in fraction:

```
>>> isinstance(Duration(3, 16), Fraction)
True
```

Durations are numeric:

```
>>> import numbers
```

```
>>> isinstance(Duration(3, 16), numbers.Number)
True
```

Durations are immutable.

Read-only properties

`Duration.`**`denominator`**

`Duration.`**`dot_count`**

New in version 2.11. Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16   1
13/16   --
14/16   2
15/16   3
16/16   0
```

Return positive integer.

Raise assignability error when duration is not assignable.

`Duration.`**`equal_or_greater_assignable`**

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
10/16   3/4
11/16   3/4
12/16   3/4
13/16   7/8
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

`Duration.`**`equal_or_greater_power_of_two`**

Equal or greater power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
```

```
3/16    1/4
4/16    1/4
5/16    1/2
6/16    1/2
7/16    1/2
8/16    1/2
9/16    1
10/16   1
11/16   1
12/16   1
13/16   1
14/16   1
15/16   1
16/16   1
```

Return new duration.

Duration.**equal_or_lesser_assignable**

Equal or lesser assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    1/4
6/16    3/8
7/16    7/16
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   3/4
13/16   3/4
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

Duration.**equal_or_lesser_power_of_two**

Equal or lesser power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/8
4/16    1/4
5/16    1/4
6/16    1/4
7/16    1/4
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   1/2
13/16   1/2
14/16   1/2
15/16   1/2
16/16   1
```

Return new duration.

Duration.**flag_count**

New in version 2.11. Nonnegative integer number of flags required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0
```

Return nonnegative integer.

Duration.**has_power_of_two_denominator**
New in version 2.11. True when duration is an integer power of `2`. Otherwise false:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration, duration.has_power_of_two_denominator)
...
1       True
1/2     True
1/3     False
1/4     True
1/5     False
1/6     False
1/7     False
1/8     True
1/9     False
1/10    False
1/11    False
1/12    False
1/13    False
1/14    False
1/15    False
1/16    True
```

Return boolean.

Duration.**imag**
Real numbers have no imaginary component.

Duration.**implied_prolation**
New in version 2.11. Implied prolation of multiplier:

```
>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)
...     print '{}\t{}'.format(multiplier, multiplier.implied_prolation)
...
1       1
1/2     1
1/3     2/3
1/4     1
1/5     4/5
1/6     2/3
1/7     4/7
1/8     1
1/9     8/9
1/10    4/5
1/11    8/11
```

```
1/12    2/3
1/13    8/13
1/14    4/7
1/15    8/15
1/16    1
```

Return new multipler.

`Duration.is_assignable`

New in version 2.11. True when assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16), duration.is_assignable)
...
0/16    False
1/16    True
2/16    True
3/16    True
4/16    True
5/16    False
6/16    True
7/16    True
8/16    True
9/16    False
10/16   False
11/16   False
12/16   True
13/16   False
14/16   True
15/16   True
16/16   True
```

Return boolean.

`Duration.lilypond_duration_string`

New in version 2.11. LilyPond duration string of assignable duration.

```
>>> Duration(3, 16).lilypond_duration_string
'8.'
```

Return string.

Raise assignability error when duration is not assignable.

`Duration.numerator`

`Duration.pair`

New in version 2.9. Read-only pair of duration numerator and denominator:

```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Return integer pair.

`Duration.prolation_string`

New in version 2.11. Prolation string.

```
>>> generator = durationtools.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
...
1       1:1
2       1:2
1/2     2:1
1/3     3:1
3       1:3
```

```
4       1:4
3/2     2:3
2/3     3:2
1/4     4:1
1/5     5:1
5       1:5
6       1:6
5/2     2:5
4/3     3:4
3/4     4:3
2/5     5:2
```

Return string.

`Duration.real`
Real numbers are their real component.

`Duration.reciprocal`
New in version 2.11. Reciprocal of duration.

Return newly constructed duration.

`Duration.storage_format`
Storage format of Abjad object.

Return string.

Methods

`Duration.conjugate()`
Conjugate is a no-op for Reals.

classmethod `Duration.from_decimal(dec)`
Converts a finite Decimal instance to a rational number, exactly.

classmethod `Duration.from_float(f)`
Converts a finite float to a rational number, exactly.

Beware that Fraction.from_float(0.3) != Fraction(3, 10).

`Duration.limit_denominator(max_denominator=1000000)`
Closest Fraction to self with denominator at most max_denominator.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

`Duration.with_denominator(denominator)`
Duration with *denominator*:

```
>>> duration = Duration(1, 4)
```

```
>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Return new duration.

Special methods

Duration.**__abs__**(**args*)

Duration.**__add__**(**args*)

Duration.**__complex__**()
complex(self) == complex(float(self), 0)

Duration.**__copy__**()

Duration.**__deepcopy__**(*memo*)

Duration.**__div__**(**args*)

Duration.**__divmod__**(**args*)

Duration.**__eq__**(*arg*)

Duration.**__float__**()
float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

Duration.**__floordiv__**(*a*, *b*)
a // b

Duration.**__ge__**(*arg*)

Duration.**__gt__**(*arg*)

Duration.**__hash__**()
hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

Duration.**__le__**(*arg*)

Duration.**__lt__**(*arg*)

Duration.**__mod__**(**args*)

Duration.**__mul__**(**args*)

Duration.**__ne__**(*arg*)

Duration.**__neg__**(**args*)

Duration.**__nonzero__**(*a*)
a != 0

Duration.**__pos__**(**args*)

Duration.**__pow__**(**args*)

Duration.**__radd__**(**args*)

Duration.**__rdiv__**(**args*)

Duration.**__rdivmod__**(**args*)

Duration.**__repr__**()

Duration.**__rfloordiv__**(*b*, *a*)
a // b

Duration.**__rmod__**(**args*)

Duration.**__rmul__**(**args*)

Duration.**__rpow__**(**args*)

Duration.**__rsub__**(**args*)

Duration.**__rtruediv__**(**args*)

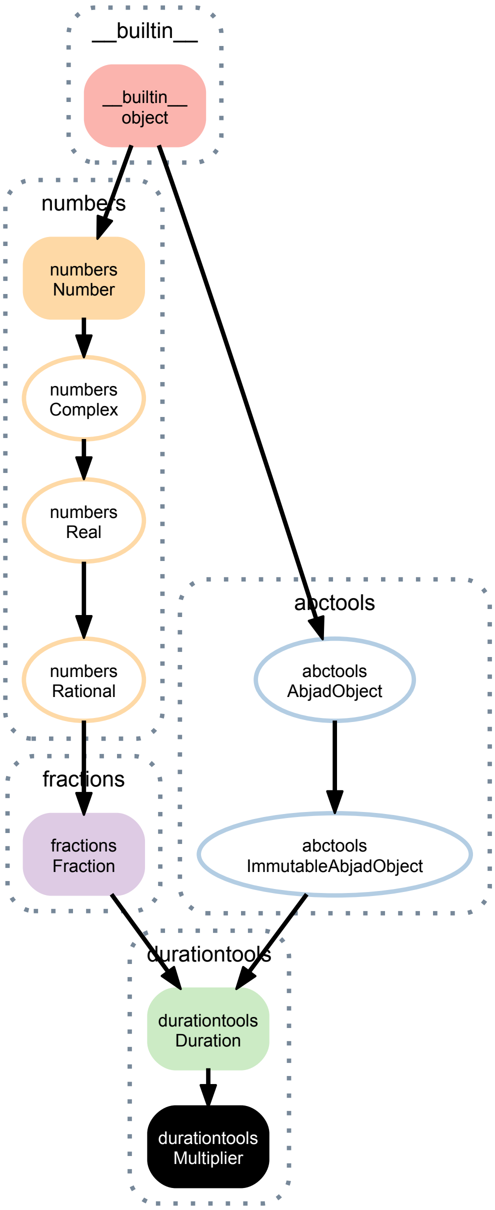
Duration.**__str__**()
str(self)

Duration.**__sub__**(**args*)

Duration.**__truediv__**(**args*)

Duration.**__trunc__**(*a*)
trunc(a)

6.1.2 durationtools.Multiplier



class durationtools.**Multiplier**(**args*, ***kwargs*)
New in version 2.11. Multiplier.

Read-only properties

Multiplier.**denominator**

Multiplier.**dot_count**

New in version 2.11. Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16   1
13/16   --
14/16   2
15/16   3
16/16   0
```

Return positive integer.

Raise assignability error when duration is not assignable.

Multiplier.**equal_or_greater_assignable**

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
10/16   3/4
11/16   3/4
12/16   3/4
13/16   7/8
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

Multiplier.**equal_or_greater_power_of_two**

Equal or greater power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/4
4/16    1/4
5/16    1/2
6/16    1/2
```

```
7/16    1/2
8/16    1/2
9/16    1
10/16   1
11/16   1
12/16   1
13/16   1
14/16   1
15/16   1
16/16   1
```

Return new duration.

`Multiplier.`**`equal_or_lesser_assignable`**

Equal or lesser assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    1/4
6/16    3/8
7/16    7/16
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   3/4
13/16   3/4
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

`Multiplier.`**`equal_or_lesser_power_of_two`**

Equal or lesser power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/8
4/16    1/4
5/16    1/4
6/16    1/4
7/16    1/4
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   1/2
13/16   1/2
14/16   1/2
15/16   1/2
16/16   1
```

Return new duration.

`Multiplier.`**`flag_count`**

New in version 2.11. Nonnegative integer number of flags required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0
```

Return nonnegative integer.

`Multiplier.`**`has_power_of_two_denominator`**

New in version 2.11. True when duration is an integer power of `2`. Otherwise false:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration, duration.has_power_of_two_denominator)
...
1       True
1/2     True
1/3     False
1/4     True
1/5     False
1/6     False
1/7     False
1/8     True
1/9     False
1/10    False
1/11    False
1/12    False
1/13    False
1/14    False
1/15    False
1/16    True
```

Return boolean.

`Multiplier.`**`imag`**

Real numbers have no imaginary component.

`Multiplier.`**`implied_prolation`**

New in version 2.11. Implied prolation of multiplier:

```
>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)
...     print '{}\t{}'.format(multiplier, multiplier.implied_prolation)
...
1       1
1/2     1
1/3     2/3
1/4     1
1/5     4/5
1/6     2/3
1/7     4/7
1/8     1
1/9     8/9
1/10    4/5
1/11    8/11
1/12    2/3
1/13    8/13
```

```
1/14    4/7
1/15    8/15
1/16    1
```

Return new multipler.

Multiplier.is_assignable

New in version 2.11. True when assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16), duration.is_assignable)
...
0/16    False
1/16    True
2/16    True
3/16    True
4/16    True
5/16    False
6/16    True
7/16    True
8/16    True
9/16    False
10/16   False
11/16   False
12/16   True
13/16   False
14/16   True
15/16   True
16/16   True
```

Return boolean.

Multiplier.is_proper_tuplet_multiplier

New in version 2.11. True when mutliplier is greater than `1/2` and less than `2`. Otherwise false:

```
>>> Multiplier(3, 2).is_proper_tuplet_multiplier
True
```

Return boolean.

Multiplier.lilypond_duration_string

New in version 2.11. LilyPond duration string of assignable duration.

```
>>> Duration(3, 16).lilypond_duration_string
'8.'
```

Return string.

Raise assignability error when duration is not assignable.

Multiplier.numerator

Multiplier.pair

New in version 2.9. Read-only pair of duration numerator and denominator:

```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Return integer pair.

Multiplier.prolation_string

New in version 2.11. Prolation string.

```
>>> generator = durationtools.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
```

```
...
1       1:1
2       1:2
1/2     2:1
1/3     3:1
3       1:3
4       1:4
3/2     2:3
2/3     3:2
1/4     4:1
1/5     5:1
5       1:5
6       1:6
5/2     2:5
4/3     3:4
3/4     4:3
2/5     5:2
```

Return string.

`Multiplier.`**`real`**
Real numbers are their real component.

`Multiplier.`**`reciprocal`**
New in version 2.11. Reciprocal of duration.

Return newly constructed duration.

`Multiplier.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`Multiplier.`**`conjugate`**`()`
Conjugate is a no-op for Reals.

classmethod `Multiplier.`**`from_decimal`**`(`*dec*`)`
Converts a finite Decimal instance to a rational number, exactly.

classmethod `Multiplier.`**`from_float`**`(`*f*`)`
Converts a finite float to a rational number, exactly.

Beware that Fraction.from_float(0.3) != Fraction(3, 10).

`Multiplier.`**`limit_denominator`**`(`*max_denominator=1000000*`)`
Closest Fraction to self with denominator at most max_denominator.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

`Multiplier.`**`with_denominator`**`(`*denominator*`)`
Duration with *denominator*:

```
>>> duration = Duration(1, 4)
```

```
>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Return new duration.

Special methods

`Multiplier.__abs__(*args)`

`Multiplier.__add__(*args)`

`Multiplier.__complex__()`
complex(self) == complex(float(self), 0)

`Multiplier.__copy__()`

`Multiplier.__deepcopy__(memo)`

`Multiplier.__div__(*args)`

`Multiplier.__divmod__(*args)`

`Multiplier.__eq__(arg)`

`Multiplier.__float__()`
float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

`Multiplier.__floordiv__(a, b)`
a // b

`Multiplier.__ge__(arg)`

`Multiplier.__gt__(arg)`

`Multiplier.__hash__()`
hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

`Multiplier.__le__(arg)`

`Multiplier.__lt__(arg)`

`Multiplier.__mod__(*args)`

`Multiplier.__mul__(*args)`

`Multiplier.__ne__(arg)`

`Multiplier.__neg__(*args)`

`Multiplier.__nonzero__(a)`
a != 0

`Multiplier.__pos__(*args)`

`Multiplier.__pow__(*args)`

`Multiplier.__radd__(*args)`

`Multiplier.__rdiv__(*args)`

`Multiplier.__rdivmod__(*args)`

`Multiplier.__repr__()`

`Multiplier.__rfloordiv__(b, a)`
a // b

`Multiplier.__rmod__(*args)`

`Multiplier.__rmul__(*args)`

`Multiplier.__rpow__(*args)`

`Multiplier.__rsub__(*args)`

Multiplier.**__rtruediv__**(**args*)

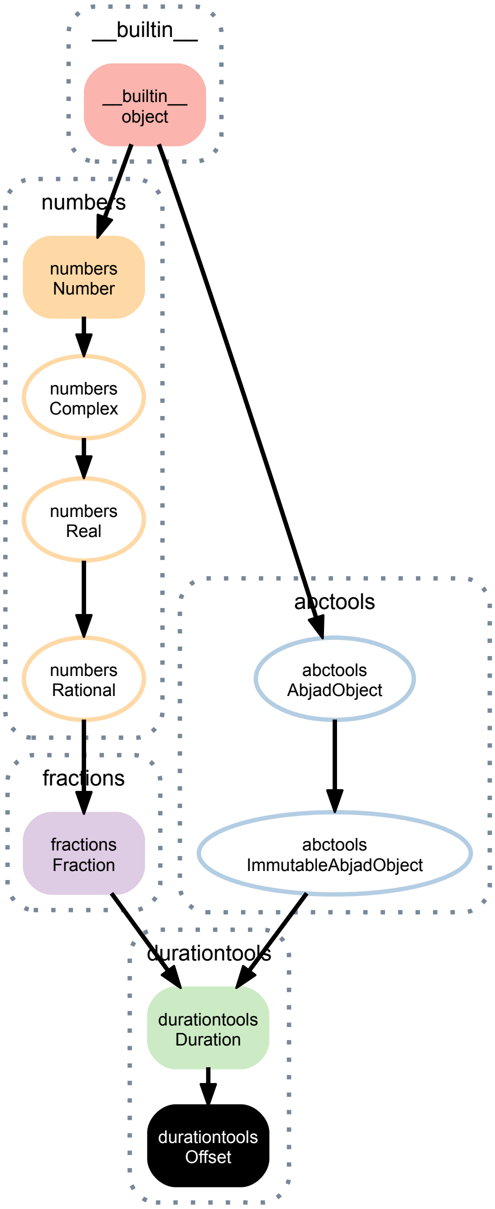
Multiplier.**__str__**()
str(self)

Multiplier.**__sub__**(**args*)

Multiplier.**__truediv__**(**args*)

Multiplier.**__trunc__**(*a*)
trunc(a)

6.1.3 durationtools.Offset



class durationtools.**Offset**(**args*, ***kwargs*)
New in version 2.0. Abjad model of offset value of musical time:

```
>>> durationtools.Offset(121, 16)
Offset(121, 16)
```

Offset inherits from duration (which inherits from built-in `Fraction`).

Read-only properties

`Offset.denominator`

`Offset.dot_count`

New in version 2.11. Positive integer number of dots required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     try:
...         duration = Duration(n, 16)
...         print '{}\t{}'.format(duration.with_denominator(16), duration.dot_count)
...     except AssignabilityError:
...         print '{}\t{}'.format(duration.with_denominator(16), '--')
...
1/16    0
2/16    0
3/16    1
4/16    0
5/16    --
6/16    1
7/16    2
8/16    0
9/16    --
10/16   --
11/16   --
12/16   1
13/16   --
14/16   2
15/16   3
16/16   0
```

Return positive integer.

Raise assignability error when duration is not assignable.

`Offset.equal_or_greater_assignable`

Equal or greater assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    3/8
6/16    3/8
7/16    7/16
8/16    1/2
9/16    3/4
10/16   3/4
11/16   3/4
12/16   3/4
13/16   7/8
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

`Offset.equal_or_greater_power_of_two`

Equal or greater power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_greater_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
```

```
3/16    1/4
4/16    1/4
5/16    1/2
6/16    1/2
7/16    1/2
8/16    1/2
9/16    1
10/16   1
11/16   1
12/16   1
13/16   1
14/16   1
15/16   1
16/16   1
```

Return new duration.

`Offset.`**`equal_or_lesser_assignable`**

Equal or lesser assignable:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_assignable
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    3/16
4/16    1/4
5/16    1/4
6/16    3/8
7/16    7/16
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   3/4
13/16   3/4
14/16   7/8
15/16   15/16
16/16   1
```

Return new duration.

`Offset.`**`equal_or_lesser_power_of_two`**

Equal or lesser power of 2:

```
>>> for numerator in range(1, 16 + 1):
...     duration = Duration(numerator, 16)
...     result = duration.equal_or_lesser_power_of_two
...     print '{}\t{}'.format(duration.with_denominator(16), result)
...
1/16    1/16
2/16    1/8
3/16    1/8
4/16    1/4
5/16    1/4
6/16    1/4
7/16    1/4
8/16    1/2
9/16    1/2
10/16   1/2
11/16   1/2
12/16   1/2
13/16   1/2
14/16   1/2
15/16   1/2
16/16   1
```

Return new duration.

`Offset.`**`flag_count`**

New in version 2.11. Nonnegative integer number of flags required to notate duration:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(n, 64)
...     print '{}\t{}'.format(duration.with_denominator(64), duration.flag_count)
...
1/64    4
2/64    3
3/64    3
4/64    2
5/64    2
6/64    2
7/64    2
8/64    1
9/64    1
10/64   1
11/64   1
12/64   1
13/64   1
14/64   1
15/64   1
16/64   0
```

Return nonnegative integer.

`Offset.has_power_of_two_denominator`

New in version 2.11. True when duration is an integer power of `2`. Otherwise false:

```
>>> for n in range(1, 16 + 1):
...     duration = Duration(1, n)
...     print '{}\t{}'.format(duration, duration.has_power_of_two_denominator)
...
1       True
1/2     True
1/3     False
1/4     True
1/5     False
1/6     False
1/7     False
1/8     True
1/9     False
1/10    False
1/11    False
1/12    False
1/13    False
1/14    False
1/15    False
1/16    True
```

Return boolean.

`Offset.imag`

Real numbers have no imaginary component.

`Offset.implied_prolation`

New in version 2.11. Implied prolation of multiplier:

```
>>> for denominator in range(1, 16 + 1):
...     multiplier = Multiplier(1, denominator)
...     print '{}\t{}'.format(multiplier, multiplier.implied_prolation)
...
1       1
1/2     1
1/3     2/3
1/4     1
1/5     4/5
1/6     2/3
1/7     4/7
1/8     1
1/9     8/9
1/10    4/5
1/11    8/11
```

```
1/12    2/3
1/13    8/13
1/14    4/7
1/15    8/15
1/16    1
```

Return new multipler.

`Offset.is_assignable`
New in version 2.11. True when assignable. Otherwise false:

```
>>> for numerator in range(0, 16 + 1):
...     duration = Duration(numerator, 16)
...     print '{}\t{}'.format(duration.with_denominator(16), duration.is_assignable)
...
0/16    False
1/16    True
2/16    True
3/16    True
4/16    True
5/16    False
6/16    True
7/16    True
8/16    True
9/16    False
10/16   False
11/16   False
12/16   True
13/16   False
14/16   True
15/16   True
16/16   True
```

Return boolean.

`Offset.lilypond_duration_string`
New in version 2.11. LilyPond duration string of assignable duration.

```
>>> Duration(3, 16).lilypond_duration_string
'8.'
```

Return string.

Raise assignability error when duration is not assignable.

`Offset.numerator`

`Offset.pair`
New in version 2.9. Read-only pair of duration numerator and denominator:

```
>>> duration = Duration(3, 16)
```

```
>>> duration.pair
(3, 16)
```

Return integer pair.

`Offset.prolation_string`
New in version 2.11. Prolation string.

```
>>> generator = durationtools.yield_durations(unique=True)
>>> for n in range(16):
...     rational = generator.next()
...     duration = Duration(rational)
...     print '{}\t{}'.format(duration, duration.prolation_string)
...
1       1:1
2       1:2
1/2     2:1
1/3     3:1
3       1:3
```

```
4       1:4
3/2     2:3
2/3     3:2
1/4     4:1
1/5     5:1
5       1:5
6       1:6
5/2     2:5
4/3     3:4
3/4     4:3
2/5     5:2
```

Return string.

Offset.**real**
Real numbers are their real component.

Offset.**reciprocal**
New in version 2.11. Reciprocal of duration.

Return newly constructed duration.

Offset.**storage_format**
Storage format of Abjad object.

Return string.

Methods

Offset.**conjugate**()
Conjugate is a no-op for Reals.

classmethod Offset.**from_decimal**(*dec*)
Converts a finite Decimal instance to a rational number, exactly.

classmethod Offset.**from_float**(*f*)
Converts a finite float to a rational number, exactly.

Beware that Fraction.from_float(0.3) != Fraction(3, 10).

Offset.**limit_denominator**(*max_denominator=1000000*)
Closest Fraction to self with denominator at most max_denominator.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

Offset.**with_denominator**(*denominator*)
Duration with *denominator*:

```
>>> duration = Duration(1, 4)
```

```
>>> for denominator in (4, 8, 16, 32):
...     print duration.with_denominator(denominator)
...
1/4
2/8
4/16
8/32
```

Return new duration.

Special methods

`Offset.__abs__(*args)`

`Offset.__add__(*args)`

`Offset.__complex__()`
complex(self) == complex(float(self), 0)

`Offset.__copy__()`

`Offset.__deepcopy__(memo)`

`Offset.__div__(*args)`

`Offset.__divmod__(*args)`

`Offset.__eq__(arg)`

`Offset.__float__()`
float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

`Offset.__floordiv__(a, b)`
a // b

`Offset.__ge__(arg)`

`Offset.__gt__(arg)`

`Offset.__hash__()`
hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

`Offset.__le__(arg)`

`Offset.__lt__(arg)`

`Offset.__mod__(*args)`

`Offset.__mul__(*args)`

`Offset.__ne__(arg)`

`Offset.__neg__(*args)`

`Offset.__nonzero__(a)`
a != 0

`Offset.__pos__(*args)`

`Offset.__pow__(*args)`

`Offset.__radd__(*args)`

`Offset.__rdiv__(*args)`

`Offset.__rdivmod__(*args)`

`Offset.__repr__()`

`Offset.__rfloordiv__(b, a)`
a // b

`Offset.__rmod__(*args)`

`Offset.__rmul__(*args)`

`Offset.__rpow__(*args)`

`Offset.__rsub__(*args)`

`Offset.__rtruediv__(*args)`

`Offset.__str__()`
str(self)

`Offset.__sub__(expr)`
New in version 2.10. Offset taken from offset returns duration:

```
>>> durationtools.Offset(2) - durationtools.Offset(1, 2)
Duration(3, 2)
```

Duration taken from offset returns another offset:

```
>>> durationtools.Offset(2) - durationtools.Duration(1, 2)
Offset(3, 2)
```

Coerce *expr* to offset when *expr* is neither offset nor duration:

```
>>> durationtools.Offset(2) - Fraction(1, 2)
Duration(3, 2)
```

Return duration or offset.

`Offset.__truediv__(*args)`

`Offset.__trunc__(a)`
trunc(a)

6.2 Functions

6.2.1 durationtools.all_are_durations

`durationtools.all_are_durations(expr)`
New in version 2.6. True when *expr* is a sequence of Abjad durations:

```
>>> durations = [Duration((3, 16)), Duration((4, 16))]
```

```
>>> durationtools.all_are_durations(durations)
True
```

True when *expr* is an empty sequence:

```
>>> durationtools.all_are_durations([])
True
```

Otherwise false:

```
>>> durationtools.all_are_durations('foo')
False
```

Return boolean.

6.2.2 durationtools.count_offsets_in_expr

`durationtools.count_offsets_in_expr(expr)`
Count offsets in *expr*.

Example 1.:

```
>>> score = Score()
>>> score.append(Staff("c'4. d'8 e'2"))
>>> score.append(Staff(r'\clef bass c4 b,4 a,2'))
>>> f(score)
\new Score <<
    \new Staff {
```

```
            c'4.
            d'8
            e'2
        }
        \new Staff {
            \clef "bass"
            c4
            b,4
            a,2
        }
>>
```

```
>>> counter = durationtools.count_offsets_in_expr(score.leaves)
>>> for offset, count in sorted(counter.items()):
...     offset, count
...
(Offset(0, 1), 2)
(Offset(1, 4), 2)
(Offset(3, 8), 2)
(Offset(1, 2), 4)
(Offset(1, 1), 2)
```

Example 2.:

```
>>> a = timespantools.Timespan(0, 10)
>>> b = timespantools.Timespan(5, 15)
>>> c = timespantools.Timespan(15, 20)
```

```
>>> counter = durationtools.count_offsets_in_expr((a, b, c))
>>> for offset, count in sorted(counter.items()):
...     offset, count
...
(Offset(0, 1), 1)
(Offset(5, 1), 1)
(Offset(10, 1), 1)
(Offset(15, 1), 2)
(Offset(20, 1), 1)
```

Return Counter.

6.2.3 durationtools.durations_to_integers

durationtools.**durations_to_integers**(*durations*)

Change *durations* to integers:

```
>>> durations = [Duration(2, 4), 3, (5, 16)]
>>> for integer in durationtools.durations_to_integers(durations):
...     integer
...
8
48
5
```

Return new object of *durations* type.

6.2.4 durationtools.durations_to_nonreduced_fractions_with_common_denominator

durationtools.**durations_to_nonreduced_fractions_with_common_denominator**(*durations*)

New in version 2.0. Change *durations* to nonreduced fractions with least common denominator:

```
>>> durations = [Duration(2, 4), 3, (5, 16)]
>>> for x in durationtools.durations_to_nonreduced_fractions_with_common_denominator(
...     durations):
...     x
...
NonreducedFraction(8, 16)
```

```
NonreducedFraction(48, 16)
NonreducedFraction(5, 16)
```

Return new object of *durations* type.

6.2.5 durationtools.group_nonreduced_fractions_by_implied_prolation

durationtools.**group_nonreduced_fractions_by_implied_prolation**(*durations*)

New in version 1.1. Group *durations* by implied prolation:

```
>>> durations = [(1, 4), (1, 8), (1, 3), (1, 6), (1, 4)]
```

```
>>> for group in durationtools.group_nonreduced_fractions_by_implied_prolation(durations):
...     group
[NonreducedFraction(1, 4), NonreducedFraction(1, 8)]
[NonreducedFraction(1, 3), NonreducedFraction(1, 6)]
[NonreducedFraction(1, 4)]
```

Return list of duration lists.

6.2.6 durationtools.numeric_seconds_to_clock_string

durationtools.**numeric_seconds_to_clock_string**(*seconds*, *escape_ticks=False*)

New in version 2.0. Example 1. Change numeric *seconds* to clock string:

```
>>> durationtools.numeric_seconds_to_clock_string(117)
'1\'57"'
```

Example 2. Change numeric *seconds* to escaped clock string:

```
>>> note = Note("c'4")
>>> clock_string = durationtools.numeric_seconds_to_clock_string(117, escape_ticks=True)
```

```
>>> markuptools.Markup('"%s"' % clock_string, Up)(note)
Markup(('1\'57\\"',), direction=Up)(c'4)
```

```
>>> f(note)
c'4 ^ \markup { 1'57\\" }
```

Return string.

6.2.7 durationtools.yield_durations

durationtools.**yield_durations**(*unique=False*)

New in version 2.0. Example 1. Yield all positive durations in Cantor diagonalized order:

```
>>> generator = durationtools.yield_durations()
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(1, 1)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(1, 2)
Duration(1, 1)
Duration(2, 1)
```

```
Duration(5, 1)
Duration(6, 1)
```

Example 2. Yield all positive durations in Cantor diagonalized order uniquely:

```
>>> generator = durationtools.yield_durations(unique=True)
>>> for n in range(16):
...     generator.next()
...
Duration(1, 1)
Duration(2, 1)
Duration(1, 2)
Duration(1, 3)
Duration(3, 1)
Duration(4, 1)
Duration(3, 2)
Duration(2, 3)
Duration(1, 4)
Duration(1, 5)
Duration(5, 1)
Duration(6, 1)
Duration(5, 2)
Duration(4, 3)
Duration(3, 4)
Duration(2, 5)
```

Return generator.

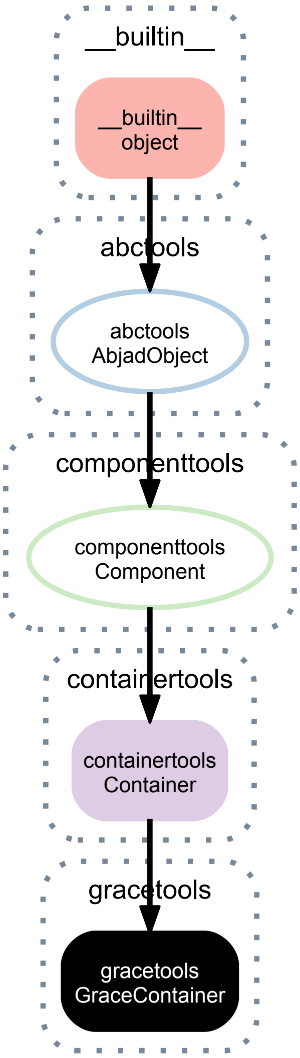
CHAPTER

SEVEN

GRACETOOLS

7.1 Concrete Classes

7.1.1 gracetools.GraceContainer



class gracetools.**GraceContainer**(*music=None*, *kind='grace'*, ***kwargs*)

Abjad model of grace music:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(voice[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> gracetools.GraceContainer(grace_notes, kind='grace')(voice[1])
Note("d'8")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    \grace {
        c'16
        d'16
    }
    d'8
    e'8
    f'8 ]
}
```

```
>>> after_grace_notes = [Note("e'16"), Note("f'16")]
>>> gracetools.GraceContainer(after_grace_notes, kind='after')(voice[1])
Note("d'8")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    \grace {
        c'16
        d'16
    }
    \afterGrace
    d'8
    {
        e'16
        f'16
    }
    e'8
    f'8 ]
}
```

Grace objects are containers you can fill with notes, rests and chords.

Grace containers override the special `__call__` method.

Use `GraceContainer()` to attach grace containers to nongrace notes, rests and chords.

Read-only properties

`GraceContainer.`**`contents_duration`**

`GraceContainer.`**`descendants`**
Read-only reference to component descendants score selection.

`GraceContainer.`**`duration`**

`GraceContainer.`**`duration_in_seconds`**

`GraceContainer.`**`leaves`**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

GraceContainer.**lilypond_format**

GraceContainer.**lineage**
Read-only reference to component lineage score selection.

GraceContainer.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

GraceContainer.**override**
Read-only reference to LilyPond grob override component plug-in.

GraceContainer.**parent**

GraceContainer.**parentage**
Read-only reference to component parentage score selection.

GraceContainer.**preprolated_duration**

GraceContainer.**prolation**

GraceContainer.**set**
Read-only reference LilyPond context setting component plug-in.

GraceContainer.**spanners**
Read-only reference to unordered set of spanners attached to component.

GraceContainer.**storage_format**
Storage format of Abjad object.

Return string.

GraceContainer.**timespan**
Read-only timespan of component.

GraceContainer.**timespan_in_seconds**
Read-only timespan of component in seconds.

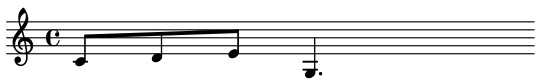
Read/write properties

GraceContainer.**is_parallel**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

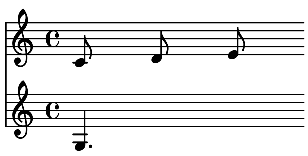
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

`GraceContainer.`**`kind`**

Get *kind* of grace container:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> gracetools.GraceContainer([Note("cs'16")], kind = 'grace')(staff[1])
Note("d'8")
>>> grace_container = staff[1].grace
>>> grace_container.kind
'grace'
```

Return string.

Set *kind* of grace container:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> gracetools.GraceContainer([Note("cs'16")], kind = 'grace')(staff[1])
Note("d'8")
>>> grace_container = staff[1].grace
>>> grace_container.kind = 'acciaccatura'
>>> grace_container.kind
'acciaccatura'
```

Set string.

Valid options include `'after'`, `'grace'`, `'acciaccatura'`, `'appoggiatura'`.

Methods

`GraceContainer.`**`append`**(*component*)

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

GraceContainer.**detach**()

Detach grace container from leaf:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> grace_container = gracetools.GraceContainer([Note("cs'16")], kind = 'grace')
>>> grace_container(staff[1])
Note("d'8")
>>> f(staff)
\new Staff {
    c'8
    \grace {
        cs'16
    }
    d'8
    e'8
    f'8
}
```

```
>>> grace_container.detach()
GraceContainer()
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

Return grace container.

GraceContainer.**extend**(*expr*)

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
```

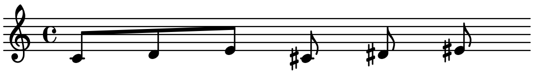
```
        e'8 ]
    }
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`GraceContainer.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`GraceContainer.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

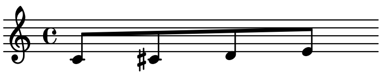
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`GraceContainer.`**`pop`**`(`*i=-1*`)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

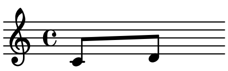
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`GraceContainer.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

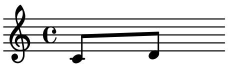
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

GraceContainer.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

GraceContainer.**__call__**(*arg*)

GraceContainer.**__contains__**(*expr*)
True if expr is in container, otherwise False.

GraceContainer.**__copy__**(**args*)

GraceContainer.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

GraceContainer.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

GraceContainer.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraceContainer.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

GraceContainer.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GraceContainer.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

GraceContainer.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

GraceContainer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraceContainer.**__len__**()
Return nonnegative integer number of components in container.

GraceContainer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraceContainer.**__mul__**(*n*)

GraceContainer.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

GraceContainer.**__radd__**(*expr*)
Extend container by contents of expr to the right.

GraceContainer.**__repr__**()

GraceContainer.**__rmul__**(*n*)

GraceContainer.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

7.2 Functions

7.2.1 gracetools.all_are_grace_containers

gracetools.**all_are_grace_containers**(*expr, kind=None*)
New in version 2.6. True when *expr* is a sequence of Abjad grace containers:

```
>>> graces = [
...     gracetools.GraceContainer("<c' e' g'>4"),
...     gracetools.GraceContainer("<c' f' a'>4")]
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> grace_container = gracetools.GraceContainer(grace_notes, kind='grace')
>>> grace_container(voice[1])
Note("d'8")
```

```
>>> f(voice)
\new Voice {
    c'8
    \grace {
        c'16
        d'16
    }
    d'8
    e'8
    f'8
}
```

```
>>> gracetools.all_are_grace_containers([grace_container])
True
```

True when *expr* is an empty sequence:

```
>>> gracetools.all_are_grace_containers([])
True
```

Otherwise false:

```
>>> gracetools.all_are_grace_containers('foo')
False
```

Return boolean.

7.2.2 gracetools.detach_grace_containers_attached_to_leaf

gracetools.**detach_grace_containers_attached_to_leaf**(*leaf, kind=None*)
New in version 2.0. Detach grace containers attached to *leaf*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> grace_container = gracetools.GraceContainer([Note("cs'16")], kind='grace')
>>> grace_container(staff[1])
Note("d'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    \grace {
        cs'16
    }
    d'8
    e'8
    f'8
}
```

```
>>> gracetools.get_grace_containers_attached_to_leaf(staff[1])
(GraceContainer(cs'16),)
```

```
>>> gracetools.detach_grace_containers_attached_to_leaf(staff[1])
(GraceContainer(),)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> gracetools.get_grace_containers_attached_to_leaf(staff[1])
()
```

Return tuple.

7.2.3 gracetools.detach_grace_containers_attached_to_leaves_in_expr

gracetools.**detach_grace_containers_attached_to_leaves_in_expr**(*expr*, *kind=None*)

New in version 2.9. Detach grace containers attached to leaves in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> grace_container = gracetools.GraceContainer([Note("cs'16")], kind='grace')
>>> grace_container(staff[1])
Note("d'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    \grace {
        cs'16
    }
    d'8
    e'8
    f'8
}
```

```
>>> gracetools.detach_grace_containers_attached_to_leaves_in_expr(staff)
(GraceContainer(),)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

Return tuple of zero or more grace containers.

7.2.4 gracetools.get_grace_containers_attached_to_leaf

gracetools.**get_grace_containers_attached_to_leaf**(*leaf*, *kind=None*)

New in version 2.0. Example 1. Get all grace containers attached to *leaf*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> gracetools.GraceContainer([Note("cs'16")], kind='grace')(staff[1])
Note("d'8")
>>> gracetools.GraceContainer([Note("ds'16")], kind='after')(staff[1])
Note("d'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    \grace {
        cs'16
    }
    \afterGrace
    d'8
    {
        ds'16
    }
    e'8
    f'8
}
```

```
>>> gracetools.get_grace_containers_attached_to_leaf(staff[1])
(GraceContainer(cs'16), GraceContainer(ds'16))
```

Example 2. Get only (proper) grace containers attached to *leaf*:

```
>>> gracetools.get_grace_containers_attached_to_leaf(staff[1], kind='grace')
(GraceContainer(cs'16),)
```

Example 3. Get only after grace containers attached to *leaf*:

```
>>> gracetools.get_grace_containers_attached_to_leaf(staff[1], kind='after')
(GraceContainer(ds'16),)
```

Return tuple.

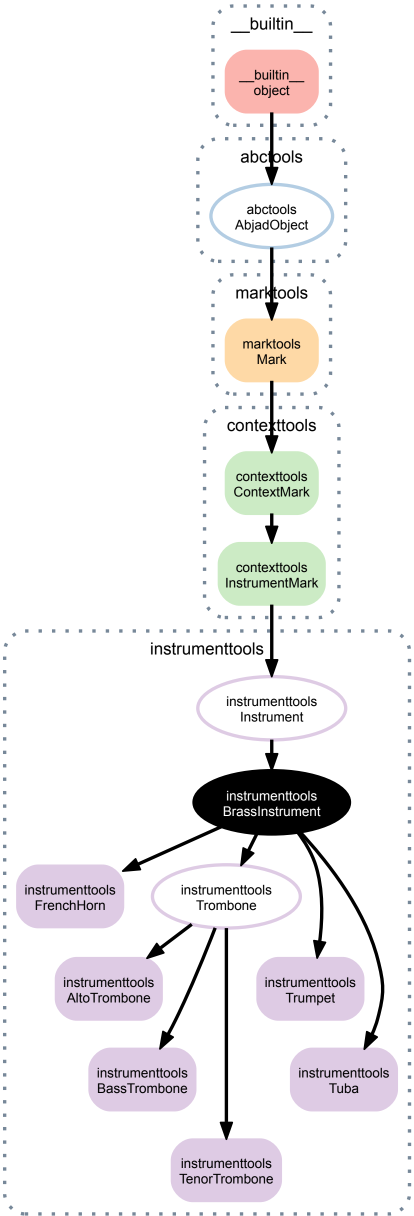
CHAPTER

EIGHT

INSTRUMENTTOOLS

8.1 Abstract Classes

8.1.1 instrumenttools.BrassInstrument



class `instrumenttools.BrassInstrument(**kwargs)`
New in version 2.0. Abjad model of brass instruments.

Read-only properties

`BrassInstrument.default_instrument_name`
Read-only default instrument name.

Return string.

`BrassInstrument.default_short_instrument_name`
Read-only default short instrument name.

Return string.

`BrassInstrument.effective_context`
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`BrassInstrument.interval_of_transposition`
Read-only interval of transposition.

Return melodic diatonic interval.

`BrassInstrument.is_primary_instrument`

`BrassInstrument.is_secondary_instrument`

`BrassInstrument.is_transposing`
True when instrument is transposing. False otherwise.

Return boolean.

`BrassInstrument.lilypond_format`
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`BrassInstrument.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`BrassInstrument.storage_format`
Storage format of Abjad object.

Return string.

`BrassInstrument.target_context`
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BrassInstrument.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

BrassInstrument.**all_clefs**

BrassInstrument.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BrassInstrument.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BrassInstrument.**pitch_range**

BrassInstrument.**primary_clefs**

BrassInstrument.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BrassInstrument.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BrassInstrument.**sounding_pitch_of_fingered_middle_c**

Methods

BrassInstrument.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BrassInstrument.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BrassInstrument.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

BrassInstrument.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

BrassInstrument.**__call__**(**args*)

BrassInstrument.**__copy__**(**args*)

BrassInstrument.**__deepcopy__**(**args*)

BrassInstrument.**__delattr__**(**args*)

BrassInstrument.**__eq__**(*arg*)

BrassInstrument.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BrassInstrument.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BrassInstrument.**__hash__**()

BrassInstrument.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

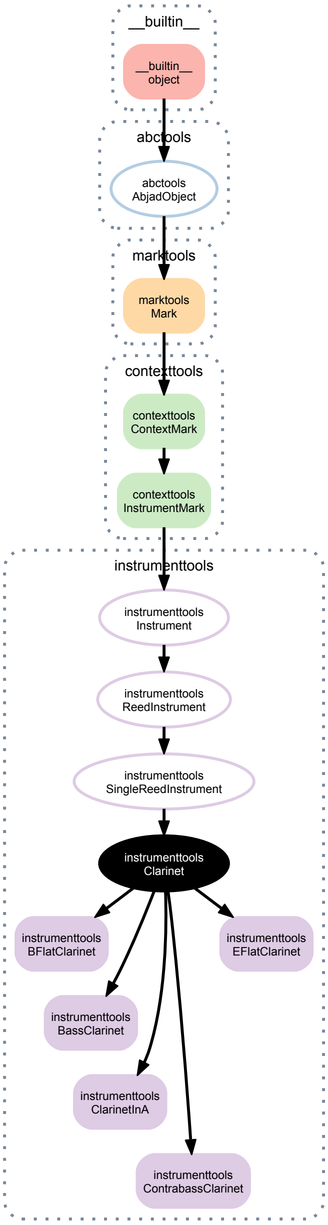
BrassInstrument.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BrassInstrument.**__ne__**(*arg*)

BrassInstrument.**__repr__**()

8.1.2 instrumenttools.Clarinet



class instrumenttools.**Clarinet**(***kwargs*)
New in version 2.6. Abjad model of the family of clarinets.

Read-only properties

Clarinet.**default_instrument_name**
Read-only default instrument name.

Return string.

Clarinet.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Clarinet.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Clarinet.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Clarinet.**is_primary_instrument**

Clarinet.**is_secondary_instrument**

Clarinet.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Clarinet.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Clarinet.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Clarinet.**storage_format**
Storage format of Abjad object.

Return string.

Clarinet.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Clarinet.`**`traditional_pitch_range`**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Clarinet.`**`all_clefs`**

`Clarinet.`**`instrument_name`**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Clarinet.`**`instrument_name_markup`**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Clarinet.`**`pitch_range`**

`Clarinet.`**`primary_clefs`**

`Clarinet.`**`short_instrument_name`**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Clarinet.`**`short_instrument_name_markup`**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Clarinet.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Clarinet.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`Clarinet.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Clarinet.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Clarinet.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`Clarinet.`**`__call__`**`(`**args*`)`

`Clarinet.`**`__copy__`**`(`**args*`)`

`Clarinet.`**`__deepcopy__`**`(`**args*`)`

`Clarinet.`**`__delattr__`**`(`**args*`)`

`Clarinet.`**`__eq__`**`(`*arg*`)`

`Clarinet.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`Clarinet.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`Clarinet.`**`__hash__`**`()`

`Clarinet.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

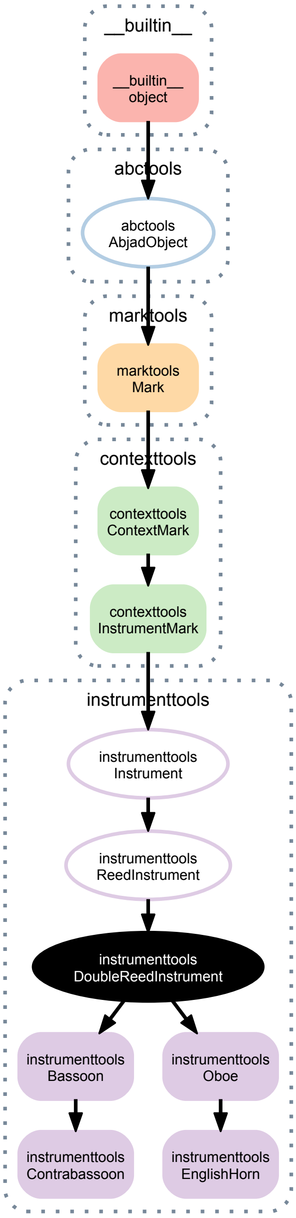
`Clarinet.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Clarinet.__ne__(arg)`

`Clarinet.__repr__()`

8.1.3 instrumenttools.DoubleReedInstrument



class `instrumenttools.DoubleReedInstrument(**kwargs)`
New in version 2.0. Abjad model of double-reed instruments.

Read-only properties

DoubleReedInstrument.**default_instrument_name**

Read-only default instrument name.

Return string.

DoubleReedInstrument.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

DoubleReedInstrument.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

DoubleReedInstrument.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

DoubleReedInstrument.**is_primary_instrument**

DoubleReedInstrument.**is_secondary_instrument**

DoubleReedInstrument.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

DoubleReedInstrument.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

DoubleReedInstrument.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

DoubleReedInstrument.**storage_format**

Storage format of Abjad object.

Return string.

DoubleReedInstrument.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

DoubleReedInstrument.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

DoubleReedInstrument.**all_clefs**

DoubleReedInstrument.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

DoubleReedInstrument.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

DoubleReedInstrument.**pitch_range**

DoubleReedInstrument.**primary_clefs**

DoubleReedInstrument.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

DoubleReedInstrument.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`DoubleReedInstrument.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`DoubleReedInstrument.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`DoubleReedInstrument.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`DoubleReedInstrument.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`DoubleReedInstrument.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`DoubleReedInstrument.`**`__call__`**`(`**args*`)`

`DoubleReedInstrument.`**`__copy__`**`(`**args*`)`

`DoubleReedInstrument.`**`__deepcopy__`**`(`**args*`)`

`DoubleReedInstrument.`**`__delattr__`**`(`**args*`)`

`DoubleReedInstrument.`**`__eq__`**`(`*arg*`)`

`DoubleReedInstrument.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DoubleReedInstrument.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`DoubleReedInstrument.`**`__hash__`**`()`

`DoubleReedInstrument.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

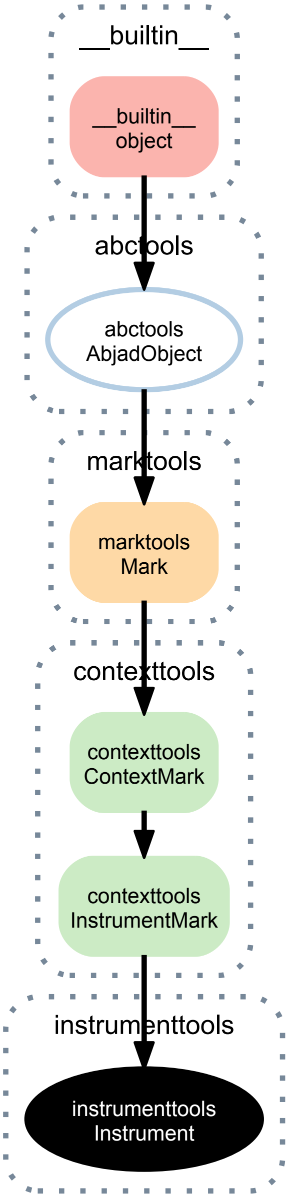
`DoubleReedInstrument.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DoubleReedInstrument.__ne__(arg)`

`DoubleReedInstrument.__repr__()`

8.1.4 instrumenttools.Instrument



class `instrumenttools.Instrument`(*instrument_name=None*, *short_instrument_name=None*, *instrument_name_markup=None*, *short_instrument_name_markup=None*, *target_context=None*)
New in version 2.0. Abjad model of the musical instrument.

Read-only properties

Instrument.**default_instrument_name**
Read-only default instrument name.

Return string.

Instrument.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Instrument.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Instrument.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Instrument.**is_primary_instrument**

Instrument.**is_secondary_instrument**

Instrument.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Instrument.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Instrument.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Instrument.**storage_format**
Storage format of Abjad object.

Return string.

Instrument.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Instrument.traditional_pitch_range`
Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Instrument.all_clefs`

`Instrument.instrument_name`
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Instrument.instrument_name_markup`
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Instrument.pitch_range`

`Instrument.primary_clefs`

`Instrument.short_instrument_name`
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Instrument.short_instrument_name_markup`
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Instrument.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Instrument.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`Instrument.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Instrument.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Instrument.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`Instrument.`**`__call__`**`(`**args*`)`

`Instrument.`**`__copy__`**`(`**args*`)`

`Instrument.`**`__deepcopy__`**`(`**args*`)`

`Instrument.`**`__delattr__`**`(`**args*`)`

`Instrument.`**`__eq__`**`(`*arg*`)`

`Instrument.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Instrument.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Instrument.`**`__hash__`**`()`

`Instrument.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

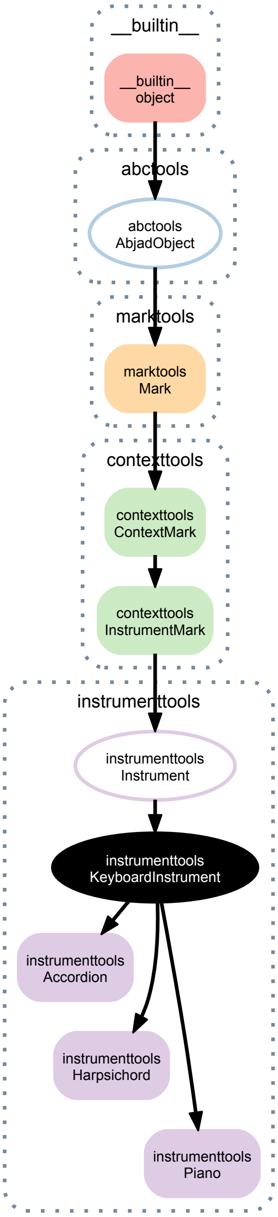
`Instrument.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Instrument.__ne__(arg)`

`Instrument.__repr__()`

8.1.5 instrumenttools.KeyboardInstrument



class `instrumenttools.KeyboardInstrument(**kwargs)`
New in version 2.0. Abjad model of keyboard instruments.

Read-only properties

`KeyboardInstrument.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`KeyboardInstrument.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`KeyboardInstrument.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`KeyboardInstrument.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`KeyboardInstrument.`**`is_primary_instrument`**

`KeyboardInstrument.`**`is_secondary_instrument`**

`KeyboardInstrument.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`KeyboardInstrument.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`KeyboardInstrument.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`KeyboardInstrument.`**`storage_format`**

Storage format of Abjad object.

Return string.

`KeyboardInstrument.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

KeyboardInstrument.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

KeyboardInstrument.**all_clefs**

KeyboardInstrument.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

KeyboardInstrument.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

KeyboardInstrument.**pitch_range**

KeyboardInstrument.**primary_clefs**

KeyboardInstrument.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

KeyboardInstrument.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`KeyboardInstrument.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`KeyboardInstrument.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`KeyboardInstrument.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`KeyboardInstrument.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`KeyboardInstrument.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`KeyboardInstrument.`**`__call__`**`(`**args*`)`

`KeyboardInstrument.`**`__copy__`**`(`**args*`)`

`KeyboardInstrument.`**`__deepcopy__`**`(`**args*`)`

`KeyboardInstrument.`**`__delattr__`**`(`**args*`)`

`KeyboardInstrument.`**`__eq__`**`(`*arg*`)`

`KeyboardInstrument.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`KeyboardInstrument.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`KeyboardInstrument.`**`__hash__`**`()`

`KeyboardInstrument.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

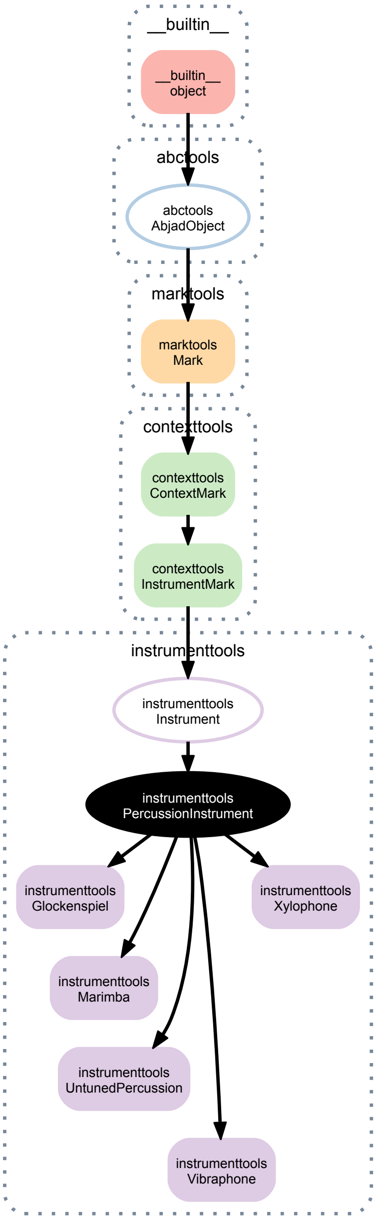
`KeyboardInstrument.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`KeyboardInstrument.__ne__(arg)`

`KeyboardInstrument.__repr__()`

8.1.6 instrumenttools.PercussionInstrument



class `instrumenttools.PercussionInstrument(**kwargs)`
New in version 2.0. Abjad model of percussion instruments.

Read-only properties

PercussionInstrument.**default_instrument_name**
Read-only default instrument name.

Return string.

PercussionInstrument.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

PercussionInstrument.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

PercussionInstrument.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

PercussionInstrument.**is_primary_instrument**

PercussionInstrument.**is_secondary_instrument**

PercussionInstrument.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

PercussionInstrument.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

PercussionInstrument.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

PercussionInstrument.**storage_format**
Storage format of Abjad object.

Return string.

PercussionInstrument.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

PercussionInstrument.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

PercussionInstrument.**all_clefs**

PercussionInstrument.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

PercussionInstrument.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

PercussionInstrument.**pitch_range**

PercussionInstrument.**primary_clefs**

PercussionInstrument.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

PercussionInstrument.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

PercussionInstrument.**sounding_pitch_of_fingered_middle_c**

Methods

PercussionInstrument.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

PercussionInstrument.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

PercussionInstrument.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

PercussionInstrument.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

PercussionInstrument.**__call__**(**args*)

PercussionInstrument.**__copy__**(**args*)

PercussionInstrument.**__deepcopy__**(**args*)

PercussionInstrument.**__delattr__**(**args*)

PercussionInstrument.**__eq__**(*arg*)

PercussionInstrument.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PercussionInstrument.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PercussionInstrument.**__hash__**()

`PercussionInstrument.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

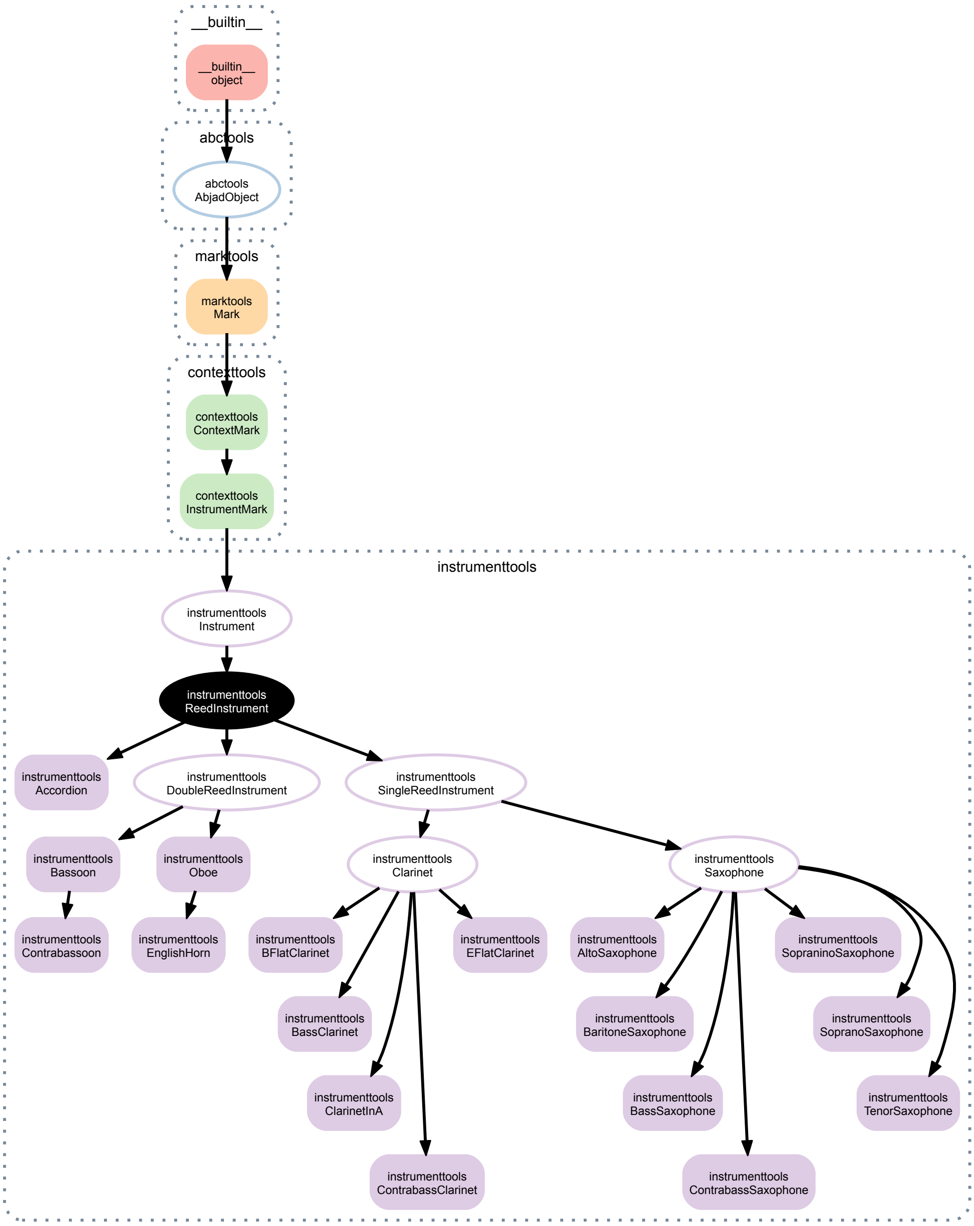
`PercussionInstrument.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PercussionInstrument.__ne__(arg)`

`PercussionInstrument.__repr__()`

8.1.7 instrumenttools.ReedInstrument



class `instrumenttools.ReedInstrument(**kwargs)`
New in version 2.0. Abjad model of reed instruments.

Read-only properties

`ReedInstrument.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`ReedInstrument.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`ReedInstrument.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`ReedInstrument.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`ReedInstrument.`**`is_primary_instrument`**

`ReedInstrument.`**`is_secondary_instrument`**

`ReedInstrument.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`ReedInstrument.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`ReedInstrument.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`ReedInstrument.`**`storage_format`**
Storage format of Abjad object.

Return string.

`ReedInstrument.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

ReedInstrument.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

ReedInstrument.**all_clefs**

ReedInstrument.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

ReedInstrument.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

ReedInstrument.**pitch_range**

ReedInstrument.**primary_clefs**

ReedInstrument.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

ReedInstrument.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

ReedInstrument.**sounding_pitch_of_fingered_middle_c**

Methods

ReedInstrument.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

ReedInstrument.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

ReedInstrument.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

ReedInstrument.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

ReedInstrument.**__call__**(**args*)

ReedInstrument.**__copy__**(**args*)

ReedInstrument.**__deepcopy__**(**args*)

ReedInstrument.**__delattr__**(**args*)

ReedInstrument.**__eq__**(*arg*)

ReedInstrument.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReedInstrument.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReedInstrument.**__hash__**()

ReedInstrument.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

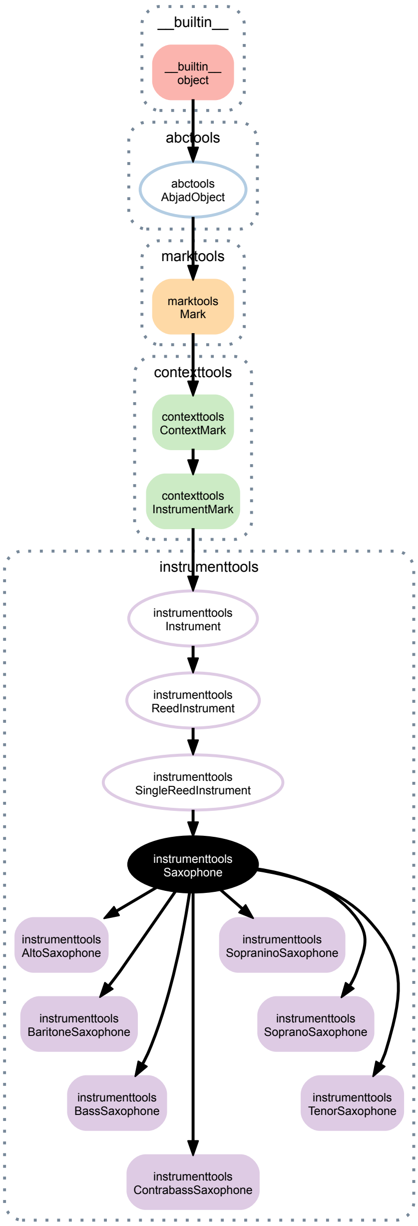
ReedInstrument.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReedInstrument.**__ne__**(*arg*)

ReedInstrument.**__repr__**()

8.1.8 instrumenttools.Saxophone



class instrumenttools.**Saxophone**(***kwargs*)
New in version 2.6. Abjad model of the family of saxophones.

Read-only properties

`Saxophone.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Saxophone.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Saxophone.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Saxophone.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Saxophone.`**`is_primary_instrument`**

`Saxophone.`**`is_secondary_instrument`**

`Saxophone.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Saxophone.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Saxophone.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Saxophone.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Saxophone.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Saxophone.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Saxophone.**all_clefs**

Saxophone.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Saxophone.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Saxophone.**pitch_range**

Saxophone.**primary_clefs**

Saxophone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Saxophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Saxophone.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Saxophone.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`Saxophone.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Saxophone.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Saxophone.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`Saxophone.`**`__call__`**`(`**args*`)`

`Saxophone.`**`__copy__`**`(`**args*`)`

`Saxophone.`**`__deepcopy__`**`(`**args*`)`

`Saxophone.`**`__delattr__`**`(`**args*`)`

`Saxophone.`**`__eq__`**`(`*arg*`)`

`Saxophone.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Saxophone.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Saxophone.`**`__hash__`**`()`

`Saxophone.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

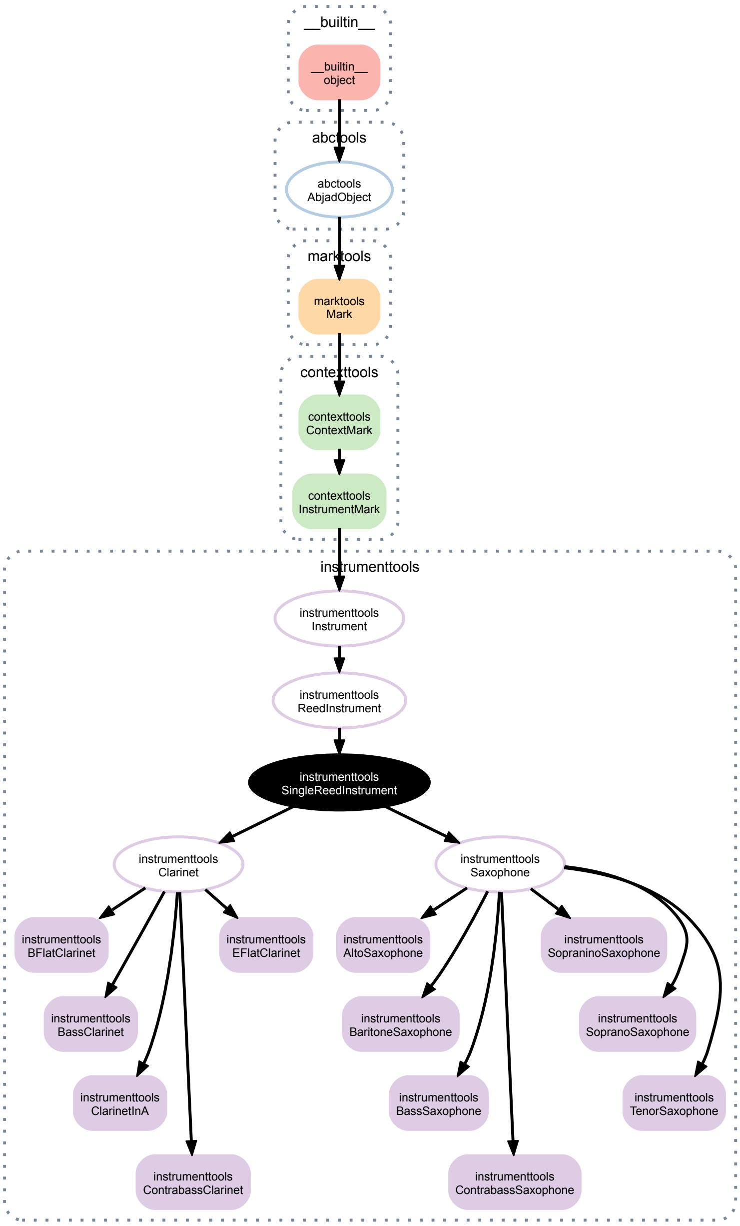
`Saxophone.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Saxophone.__ne__(arg)`

`Saxophone.__repr__()`

8.1.9 instrumenttools.SingleReedInstrument



class `instrumenttools.SingleReedInstrument(**kwargs)`
New in version 2.0. Abjad model of single-reed instruments.

Read-only properties

SingleReedInstrument.**default_instrument_name**
Read-only default instrument name.

Return string.

SingleReedInstrument.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

SingleReedInstrument.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

SingleReedInstrument.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

SingleReedInstrument.**is_primary_instrument**

SingleReedInstrument.**is_secondary_instrument**

SingleReedInstrument.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

SingleReedInstrument.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

SingleReedInstrument.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

SingleReedInstrument.**storage_format**
Storage format of Abjad object.

Return string.

SingleReedInstrument.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

SingleReedInstrument.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

SingleReedInstrument.**all_clefs**

SingleReedInstrument.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

SingleReedInstrument.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

SingleReedInstrument.**pitch_range**

SingleReedInstrument.**primary_clefs**

SingleReedInstrument.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

SingleReedInstrument.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`SingleReedInstrument.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`SingleReedInstrument.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`SingleReedInstrument.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`SingleReedInstrument.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`SingleReedInstrument.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`SingleReedInstrument.`**`__call__`**`(`**args*`)`

`SingleReedInstrument.`**`__copy__`**`(`**args*`)`

`SingleReedInstrument.`**`__deepcopy__`**`(`**args*`)`

`SingleReedInstrument.`**`__delattr__`**`(`**args*`)`

`SingleReedInstrument.`**`__eq__`**`(`*arg*`)`

`SingleReedInstrument.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`SingleReedInstrument.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`SingleReedInstrument.`**`__hash__`**`()`

`SingleReedInstrument.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

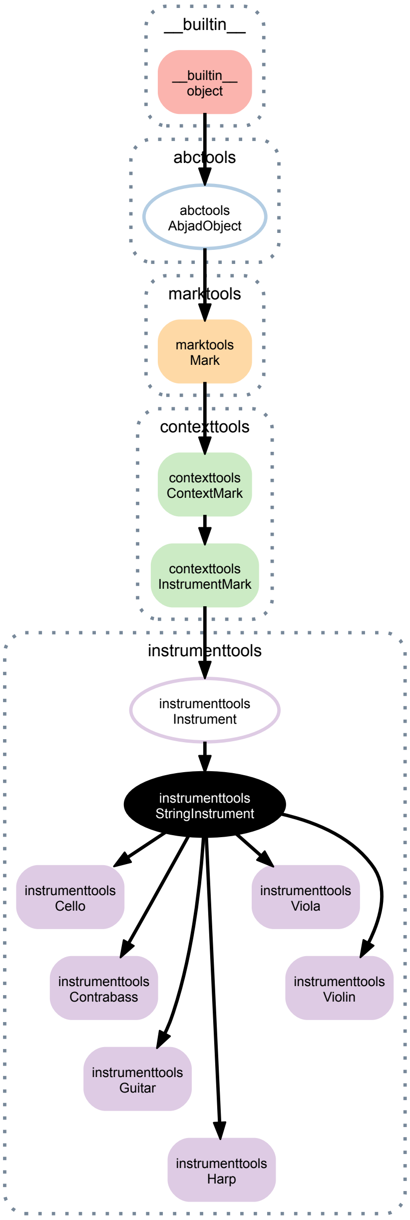
`SingleReedInstrument.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`SingleReedInstrument.`**`__ne__`**`(`*arg*`)`

`SingleReedInstrument.`**`__repr__`**`()`

8.1.10 instrumenttools.StringInstrument



class `instrumenttools.`**`StringInstrument`**`(`***kwargs*`)`
New in version 2.0. Abjad model of string instruments.

Read-only properties

`StringInstrument.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`StringInstrument.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`StringInstrument.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`StringInstrument.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`StringInstrument.`**`is_primary_instrument`**

`StringInstrument.`**`is_secondary_instrument`**

`StringInstrument.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`StringInstrument.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`StringInstrument.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`StringInstrument.`**`storage_format`**
Storage format of Abjad object.

Return string.

`StringInstrument.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

StringInstrument.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

StringInstrument.**all_clefs**

StringInstrument.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

StringInstrument.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

StringInstrument.**pitch_range**

StringInstrument.**primary_clefs**

StringInstrument.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

StringInstrument.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

StringInstrument.**sounding_pitch_of_fingered_middle_c**

Methods

StringInstrument.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

StringInstrument.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

StringInstrument.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

StringInstrument.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

StringInstrument.**__call__**(**args*)

StringInstrument.**__copy__**(**args*)

StringInstrument.**__deepcopy__**(**args*)

StringInstrument.**__delattr__**(**args*)

StringInstrument.**__eq__**(*arg*)

StringInstrument.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StringInstrument.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

StringInstrument.**__hash__**()

StringInstrument.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

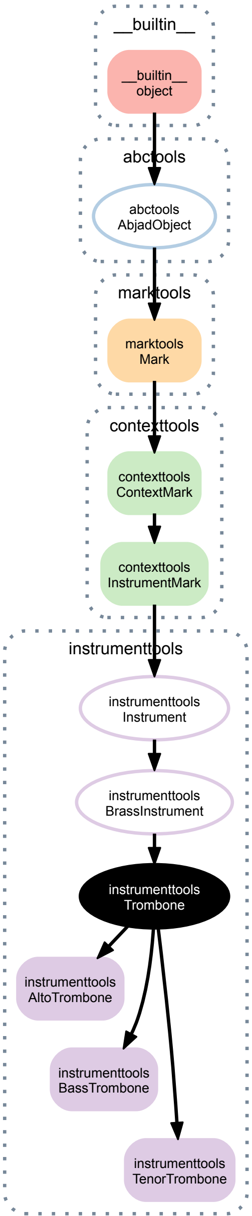
StringInstrument.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StringInstrument.**__ne__**(*arg*)

StringInstrument.**__repr__**()

8.1.11 instrumenttools.Trombone



class instrumenttools.**Trombone**(***kwargs*)
New in version 2.0. Abjad model of the family of trombones.

Read-only properties

`Trombone.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Trombone.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Trombone.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Trombone.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Trombone.`**`is_primary_instrument`**

`Trombone.`**`is_secondary_instrument`**

`Trombone.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Trombone.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Trombone.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Trombone.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Trombone.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Trombone.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Trombone.`**`all_clefs`**

`Trombone.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Trombone.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Trombone.`**`pitch_range`**

`Trombone.`**`primary_clefs`**

`Trombone.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Trombone.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Trombone.**sounding_pitch_of_fingered_middle_c**

Methods

Trombone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Trombone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Trombone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Trombone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Trombone.**__call__**(**args*)

Trombone.**__copy__**(**args*)

Trombone.**__deepcopy__**(**args*)

Trombone.**__delattr__**(**args*)

Trombone.**__eq__**(*arg*)

Trombone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Trombone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Trombone.**__hash__**()

Trombone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

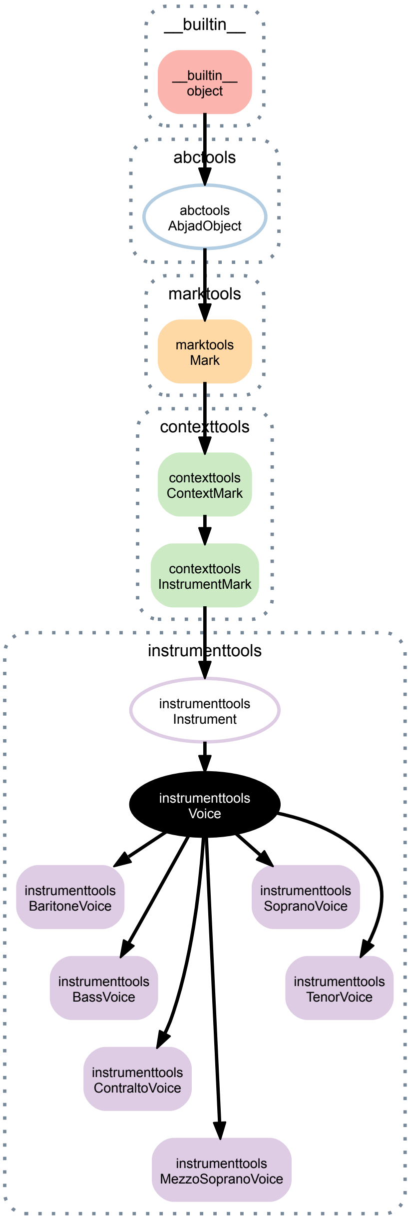
Trombone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Trombone.**__ne__**(*arg*)

Trombone.**__repr__**()

8.1.12 instrumenttools.Voice



class instrumenttools.**Voice**(***kwargs*)
New in version 2.8. Abjad model of the human voice.

Read-only properties

`Voice.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`Voice.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`Voice.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Voice.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`Voice.`**`is_primary_instrument`**

`Voice.`**`is_secondary_instrument`**

`Voice.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`Voice.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Voice.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Voice.`**`storage_format`**

Storage format of Abjad object.

Return string.

`Voice.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Voice.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Voice.**all_clefs**

Voice.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Voice.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Voice.**pitch_range**

Voice.**primary_clefs**

Voice.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Voice.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Voice.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Voice.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`Voice.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Voice.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Voice.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`Voice.`**`__call__`**`(`**args*`)`

`Voice.`**`__copy__`**`(`**args*`)`

`Voice.`**`__deepcopy__`**`(`**args*`)`

`Voice.`**`__delattr__`**`(`**args*`)`

`Voice.`**`__eq__`**`(`*arg*`)`

`Voice.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Voice.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Voice.`**`__hash__`**`()`

Voice.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

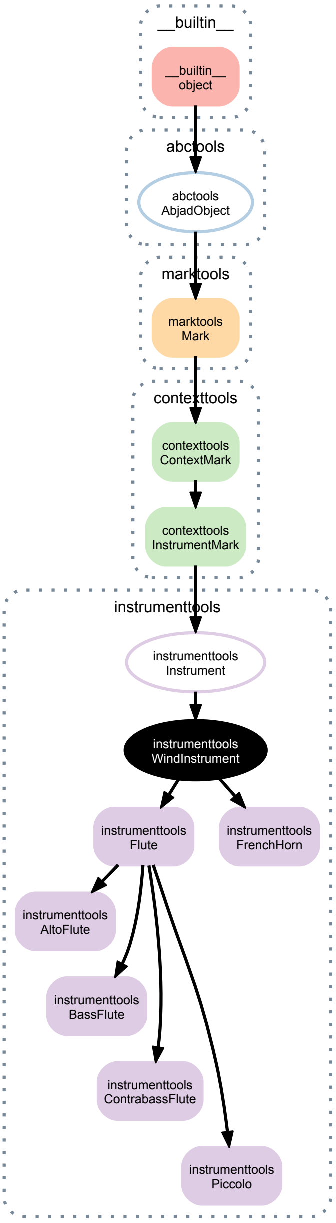
Voice.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Voice.**__ne__**(*arg*)

Voice.**__repr__**()

8.1.13 instrumenttools.WindInstrument



class instrumenttools.**WindInstrument**(***kwargs*)
New in version 2.0. Abjad model of the wind instrument.

Read-only properties

`WindInstrument.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`WindInstrument.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`WindInstrument.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`WindInstrument.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`WindInstrument.`**`is_primary_instrument`**

`WindInstrument.`**`is_secondary_instrument`**

`WindInstrument.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`WindInstrument.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`WindInstrument.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`WindInstrument.`**`storage_format`**

Storage format of Abjad object.

Return string.

`WindInstrument.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

WindInstrument.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

WindInstrument.**all_clefs**

WindInstrument.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

WindInstrument.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

WindInstrument.**pitch_range**

WindInstrument.**primary_clefs**

WindInstrument.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

WindInstrument.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`WindInstrument.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`WindInstrument.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`WindInstrument.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`WindInstrument.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`WindInstrument.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`WindInstrument.`**`__call__`**`(`**args*`)`

`WindInstrument.`**`__copy__`**`(`**args*`)`

`WindInstrument.`**`__deepcopy__`**`(`**args*`)`

`WindInstrument.`**`__delattr__`**`(`**args*`)`

`WindInstrument.`**`__eq__`**`(`*arg*`)`

`WindInstrument.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`WindInstrument.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`WindInstrument.`**`__hash__`**`()`

`WindInstrument.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`WindInstrument.__lt__(expr)`
Abjad objects by default do not implement this method.

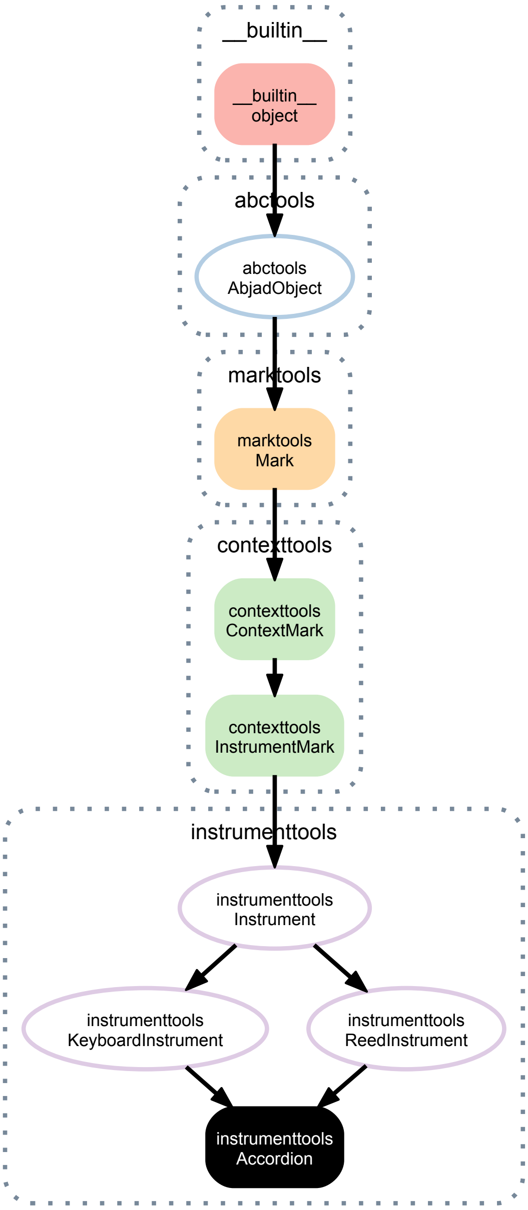
Raise exception.

`WindInstrument.__ne__(arg)`

`WindInstrument.__repr__()`

8.2 Concrete Classes

8.2.1 instrumenttools.Accordion



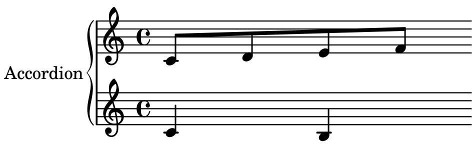
class `instrumenttools.Accordion`(*target_context=None*, ***kwargs*)
Abjad model of the accordion:

```
>>> piano_staff = scoretools.PianoStaff([Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Accordion()(piano_staff)
Accordion()(PianoStaff<<2>>)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \set PianoStaff.instrumentName = \markup { Accordion }
    \set PianoStaff.shortInstrumentName = \markup { Acc. }
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        b4
    }
>>
```

```
>>> show(piano_staff)
```



The accordion targets piano staff context by default.

Read-only properties

Accordion.**default_instrument_name**
Read-only default instrument name.

Return string.

Accordion.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Accordion.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Accordion.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Accordion.**is_primary_instrument**

Accordion.**is_secondary_instrument**

Accordion.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

`Accordion.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Accordion.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Accordion.`**`storage_format`**

Storage format of Abjad object.

Return string.

`Accordion.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Accordion.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Accordion.`**`all_clefs`**

`Accordion.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Accordion.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Accordion.`**`pitch_range`**

`Accordion.`**`primary_clefs`**

`Accordion.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Accordion.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Accordion.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Accordion.`**`attach`**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

`Accordion.`**`detach`**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Accordion.`**`get_default_performer_name`**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Accordion.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Accordion.**__call__**(**args*)

Accordion.**__copy__**(**args*)

Accordion.**__deepcopy__**(**args*)

Accordion.**__delattr__**(**args*)

Accordion.**__eq__**(*arg*)

Accordion.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Accordion.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Accordion.**__hash__**()

Accordion.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

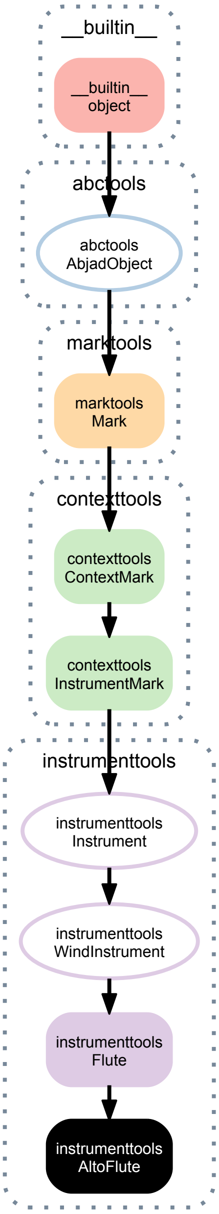
Accordion.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Accordion.**__ne__**(*arg*)

Accordion.**__repr__**()

8.2.2 instrumenttools.AltoFlute



class `instrumenttools.AltoFlute`(***kwargs*)

Abjad model of the alto flute:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.AltoFlute()(staff)
AltoFlute()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Alto flute }
    \set Staff.shortInstrumentName = \markup { Alt. fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The alto flute targets staff context by default.

Read-only properties

AltoFlute.**default_instrument_name**
Read-only default instrument name.

Return string.

AltoFlute.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

AltoFlute.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

AltoFlute.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

AltoFlute.**is_primary_instrument**

AltoFlute.**is_secondary_instrument**

AltoFlute.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

AltoFlute.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

AltoFlute.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

AltoFlute.**storage_format**
Storage format of Abjad object.

Return string.

AltoFlute.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

AltoFlute.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

AltoFlute.**all_clefs**

AltoFlute.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

AltoFlute.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

AltoFlute.**pitch_range**

AltoFlute.**primary_clefs**

AltoFlute.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

AltoFlute.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

AltoFlute.**sounding_pitch_of_fingered_middle_c**

Methods

AltoFlute.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

AltoFlute.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

AltoFlute.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

AltoFlute.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

AltoFlute.**__call__**(**args*)

AltoFlute.**__copy__**(**args*)

AltoFlute.**__deepcopy__**(**args*)

AltoFlute.**__delattr__**(**args*)

AltoFlute.**__eq__**(*arg*)

AltoFlute.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

AltoFlute.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AltoFlute.**__hash__**()

AltoFlute.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

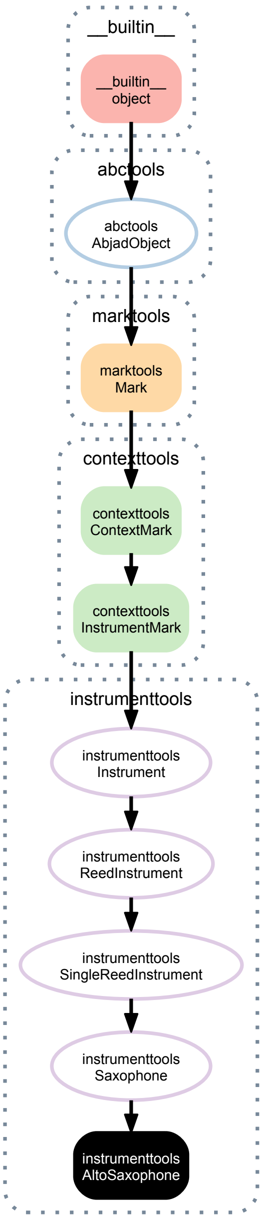
AltoFlute.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AltoFlute.**__ne__**(*arg*)

AltoFlute.**__repr__**()

8.2.3 instrumenttools.AltoSaxophone



class `instrumenttools.`**`AltoSaxophone`**`(**kwargs)`

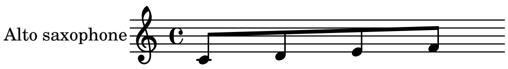
New in version 2.6. Abjad model of the alto saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.AltoSaxophone()(staff)
AltoSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Alto saxophone }
    \set Staff.shortInstrumentName = \markup { Alto sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



The alto saxophone is pitched in E-flat.

The alto saxophone targets staff context by default.

Read-only properties

`AltoSaxophone.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`AltoSaxophone.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`AltoSaxophone.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`AltoSaxophone.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`AltoSaxophone.`**`is_primary_instrument`**

`AltoSaxophone.`**`is_secondary_instrument`**

`AltoSaxophone.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`AltoSaxophone.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

AltoSaxophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

AltoSaxophone.**storage_format**
Storage format of Abjad object.

Return string.

AltoSaxophone.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

AltoSaxophone.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

AltoSaxophone.**all_clefs**

AltoSaxophone.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

AltoSaxophone.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`AltoSaxophone.`**`pitch_range`**

`AltoSaxophone.`**`primary_clefs`**

`AltoSaxophone.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`AltoSaxophone.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`AltoSaxophone.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`AltoSaxophone.`**`attach`**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

`AltoSaxophone.`**`detach`**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`AltoSaxophone.`**`get_default_performer_name`**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

AltoSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

AltoSaxophone.**__call__**(**args*)

AltoSaxophone.**__copy__**(**args*)

AltoSaxophone.**__deepcopy__**(**args*)

AltoSaxophone.**__delattr__**(**args*)

AltoSaxophone.**__eq__**(*arg*)

AltoSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AltoSaxophone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AltoSaxophone.**__hash__**()

AltoSaxophone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

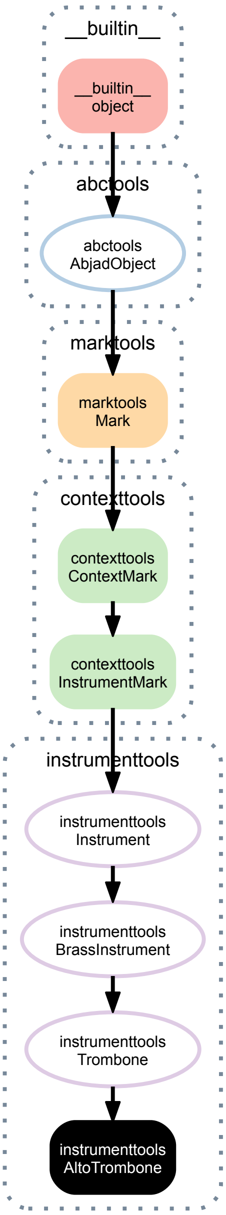
AltoSaxophone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AltoSaxophone.**__ne__**(*arg*)

AltoSaxophone.**__repr__**()

8.2.4 instrumenttools.AltoTrombone



class `instrumenttools.`**`AltoTrombone`**(***kwargs*)

New in version 2.0. Abjad model of the alto trombone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.AltoTrombone()(staff)
AltoTrombone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Alto trombone }
    \set Staff.shortInstrumentName = \markup { Alt. trb. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The tenor trombone targets staff context by default.

Read-only properties

AltoTrombone.**default_instrument_name**
Read-only default instrument name.

Return string.

AltoTrombone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

AltoTrombone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

AltoTrombone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

AltoTrombone.**is_primary_instrument**

AltoTrombone.**is_secondary_instrument**

AltoTrombone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

AltoTrombone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

AltoTrombone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

AltoTrombone.**storage_format**
Storage format of Abjad object.

Return string.

AltoTrombone.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

AltoTrombone.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

AltoTrombone.**all_clefs**

AltoTrombone.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

AltoTrombone.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

AltoTrombone.**pitch_range**

AltoTrombone.**primary_clefs**

AltoTrombone.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

AltoTrombone.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

AltoTrombone.**sounding_pitch_of_fingered_middle_c**

Methods

AltoTrombone.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

AltoTrombone.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

AltoTrombone.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

AltoTrombone.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

AltoTrombone.**__call__**(**args*)

AltoTrombone.**__copy__**(**args*)

AltoTrombone.**__deepcopy__**(**args*)

AltoTrombone.**__delattr__**(**args*)

AltoTrombone.**__eq__**(*arg*)

AltoTrombone.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`AltoTrombone.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`AltoTrombone.__hash__()`

`AltoTrombone.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

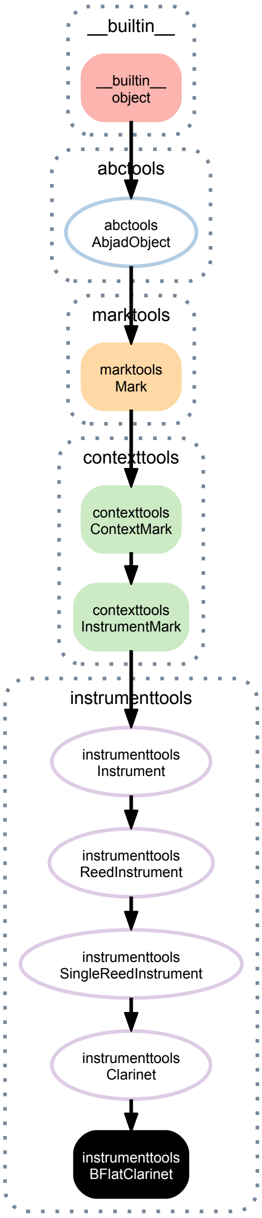
`AltoTrombone.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`AltoTrombone.__ne__(arg)`

`AltoTrombone.__repr__()`

8.2.5 instrumenttools.BFlatClarinet



class instrumenttools.**BFlatClarinet**(***kwargs*)

New in version 2.0. Abjad model of the B-flat clarinet:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.BFlatClarinet()(staff)
BFlatClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in B-flat }
    \set Staff.shortInstrumentName = \markup { Cl. in B-flat }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The B-flat clarinet targets staff context by default.

Read-only properties

BFlatClarinet.**default_instrument_name**

Read-only default instrument name.

Return string.

BFlatClarinet.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

BFlatClarinet.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

BFlatClarinet.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

BFlatClarinet.**is_primary_instrument**

BFlatClarinet.**is_secondary_instrument**

BFlatClarinet.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

BFlatClarinet.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`BFlatClarinet.start_component`

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`BFlatClarinet.storage_format`

Storage format of Abjad object.

Return string.

`BFlatClarinet.target_context`

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`BFlatClarinet.traditional_pitch_range`

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`BFlatClarinet.all_clefs`

`BFlatClarinet.instrument_name`

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`BFlatClarinet.instrument_name_markup`

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`BFlatClarinet.pitch_range`

BFlatClarinet.**primary_clefs**

BFlatClarinet.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BFlatClarinet.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BFlatClarinet.**sounding_pitch_of_fingered_middle_c**

Methods

BFlatClarinet.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

BFlatClarinet.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BFlatClarinet.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

BFlatClarinet.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

`BFlatClarinet.__call__(*args)`

`BFlatClarinet.__copy__(*args)`

`BFlatClarinet.__deepcopy__(*args)`

`BFlatClarinet.__delattr__(*args)`

`BFlatClarinet.__eq__(arg)`

`BFlatClarinet.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BFlatClarinet.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BFlatClarinet.__hash__()`

`BFlatClarinet.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

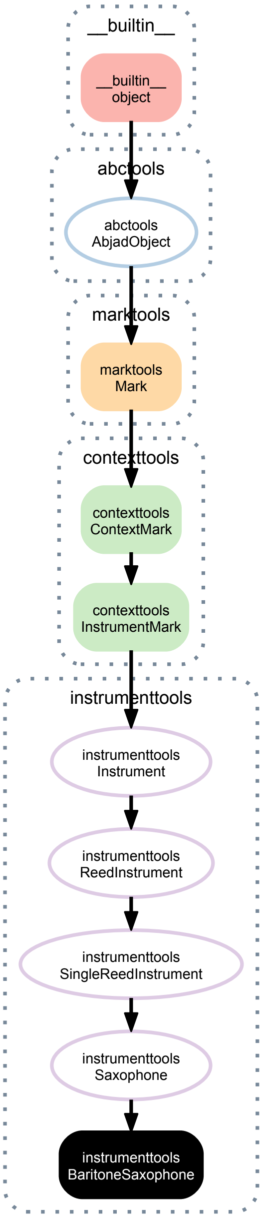
`BFlatClarinet.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BFlatClarinet.__ne__(arg)`

`BFlatClarinet.__repr__()`

8.2.6 instrumenttools.BaritoneSaxophone



class `instrumenttools.`**`BaritoneSaxophone`**(***kwargs*)

New in version 2.6. Abjad model of the baritone saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.BaritoneSaxophone()(staff)
BaritoneSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Baritone saxophone }
    \set Staff.shortInstrumentName = \markup { Bar. sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The baritone saxophone is pitched in E-flat.

The baritone saxophone targets staff context by default.

Read-only properties

BaritoneSaxophone.**default_instrument_name**
Read-only default instrument name.

Return string.

BaritoneSaxophone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

BaritoneSaxophone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

BaritoneSaxophone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

BaritoneSaxophone.**is_primary_instrument**

BaritoneSaxophone.**is_secondary_instrument**

BaritoneSaxophone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

BaritoneSaxophone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BaritoneSaxophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BaritoneSaxophone.**storage_format**
Storage format of Abjad object.

Return string.

BaritoneSaxophone.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BaritoneSaxophone.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

BaritoneSaxophone.**all_clefs**

BaritoneSaxophone.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BaritoneSaxophone.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BaritoneSaxophone.**pitch_range**

BaritoneSaxophone.**primary_clefs**

BaritoneSaxophone.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BaritoneSaxophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BaritoneSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

BaritoneSaxophone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BaritoneSaxophone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BaritoneSaxophone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

BaritoneSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

BaritoneSaxophone.**__call__**(**args*)

BaritoneSaxophone.**__copy__**(**args*)

BaritoneSaxophone.**__deepcopy__**(**args*)

BaritoneSaxophone.**__delattr__**(**args*)

BaritoneSaxophone.**__eq__**(*arg*)

BaritoneSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`BaritoneSaxophone.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BaritoneSaxophone.__hash__()`

`BaritoneSaxophone.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

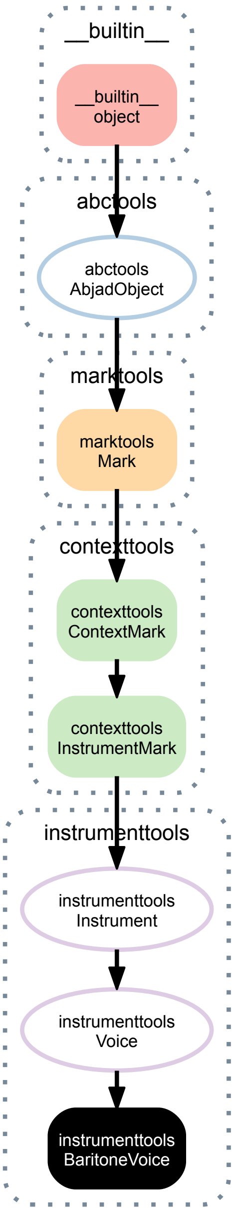
`BaritoneSaxophone.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BaritoneSaxophone.__ne__(arg)`

`BaritoneSaxophone.__repr__()`

8.2.7 instrumenttools.BaritoneVoice



class `instrumenttools.`**`BaritoneVoice`**(***kwargs*)

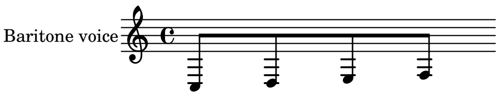
New in version 2.8. Abjad model of the baritone voice:

```
>>> staff = Staff("c8 d8 e8 f8")
```

```
>>> instrumenttools.BaritoneVoice()(staff)
BaritoneVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Baritone voice }
    \set Staff.shortInstrumentName = \markup { Baritone }
    c8
    d8
    e8
    f8
}
```

```
>>> show(staff)
```



The baritone voice targets staff context by default.

Read-only properties

`BaritoneVoice.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`BaritoneVoice.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`BaritoneVoice.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`BaritoneVoice.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`BaritoneVoice.`**`is_primary_instrument`**

`BaritoneVoice.`**`is_secondary_instrument`**

`BaritoneVoice.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`BaritoneVoice.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`BaritoneVoice.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`BaritoneVoice.`**`storage_format`**

Storage format of Abjad object.

Return string.

`BaritoneVoice.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`BaritoneVoice.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`BaritoneVoice.`**`all_clefs`**

`BaritoneVoice.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`BaritoneVoice.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BaritoneVoice.**pitch_range**

BaritoneVoice.**primary_clefs**

BaritoneVoice.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BaritoneVoice.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BaritoneVoice.**sounding_pitch_of_fingered_middle_c**

Methods

BaritoneVoice.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BaritoneVoice.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BaritoneVoice.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

BaritoneVoice.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

BaritoneVoice.**__call__**(**args*)

BaritoneVoice.**__copy__**(**args*)

BaritoneVoice.**__deepcopy__**(**args*)

BaritoneVoice.**__delattr__**(**args*)

BaritoneVoice.**__eq__**(*arg*)

BaritoneVoice.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BaritoneVoice.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BaritoneVoice.**__hash__**()

BaritoneVoice.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

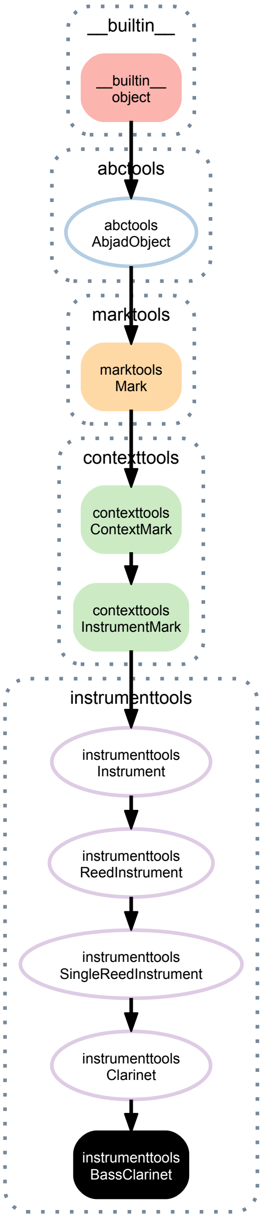
BaritoneVoice.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BaritoneVoice.**__ne__**(*arg*)

BaritoneVoice.**__repr__**()

8.2.8 instrumenttools.BassClarinet



class `instrumenttools.`**`BassClarinet`**(***kwargs*)

New in version 2.0. Abjad model of the bass clarinet:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.BassClarinet()(staff)
BassClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Bass clarinet }
    \set Staff.shortInstrumentName = \markup { Bass cl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The bass clarinet targets staff context by default.

Read-only properties

BassClarinet.**default_instrument_name**

Read-only default instrument name.

Return string.

BassClarinet.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

BassClarinet.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

BassClarinet.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

BassClarinet.**is_primary_instrument**

BassClarinet.**is_secondary_instrument**

BassClarinet.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

BassClarinet.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BassClarinet.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BassClarinet.**storage_format**

Storage format of Abjad object.

Return string.

BassClarinet.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BassClarinet.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

BassClarinet.**all_clefs**

BassClarinet.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BassClarinet.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BassClarinet.**pitch_range**

BassClarinet.**primary_clefs**

BassClarinet.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`BassClarinet.`**`short_instrument_name_markup`**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`BassClarinet.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`BassClarinet.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`BassClarinet.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`BassClarinet.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`BassClarinet.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`BassClarinet.`**`__call__`**`(`**args*`)`

`BassClarinet.`**`__copy__`**`(`**args*`)`

`BassClarinet.`**`__deepcopy__`**`(`**args*`)`

`BassClarinet.`**`__delattr__`**`(`**args*`)`

`BassClarinet.`**`__eq__`**`(`*arg*`)`

`BassClarinet.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`BassClarinet.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BassClarinet.__hash__()`

`BassClarinet.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

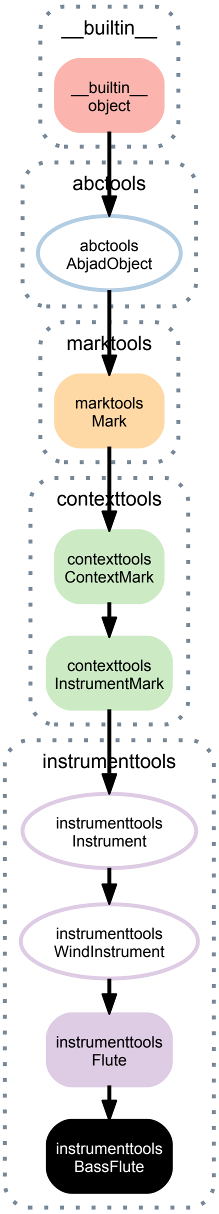
`BassClarinet.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassClarinet.__ne__(arg)`

`BassClarinet.__repr__()`

8.2.9 instrumenttools.BassFlute



class `instrumenttools.`**`BassFlute`**`(`***kwargs*`)`

New in version 2.0. Abjad model of the bass flute:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.BassFlute()(staff)
BassFlute()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Bass flute }
    \set Staff.shortInstrumentName = \markup { Bass fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The bass flute targets staff context by default.

Read-only properties

`BassFlute.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`BassFlute.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`BassFlute.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`BassFlute.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`BassFlute.`**`is_primary_instrument`**

`BassFlute.`**`is_secondary_instrument`**

`BassFlute.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`BassFlute.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BassFlute.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BassFlute.**storage_format**

Storage format of Abjad object.

Return string.

BassFlute.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BassFlute.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

BassFlute.**all_clefs**

BassFlute.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BassFlute.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BassFlute.**pitch_range**

BassFlute.**primary_clefs**

BassFlute.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BassFlute.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BassFlute.**sounding_pitch_of_fingered_middle_c**

Methods

BassFlute.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BassFlute.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BassFlute.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

BassFlute.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

`BassFlute.__call__(*args)`

`BassFlute.__copy__(*args)`

`BassFlute.__deepcopy__(*args)`

`BassFlute.__delattr__(*args)`

`BassFlute.__eq__(arg)`

`BassFlute.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassFlute.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BassFlute.__hash__()`

`BassFlute.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

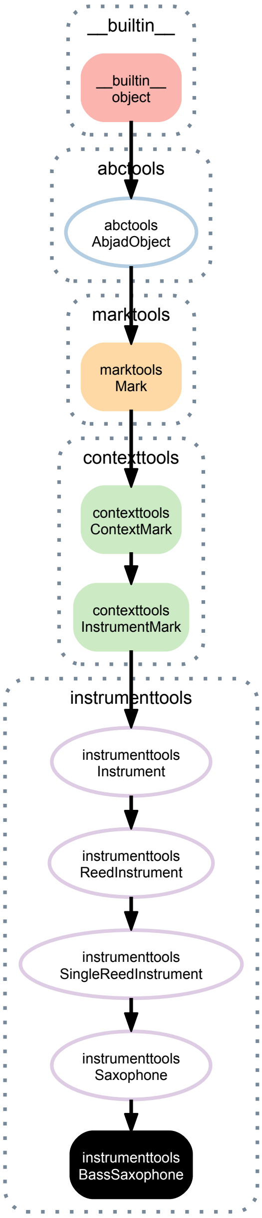
`BassFlute.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassFlute.__ne__(arg)`

`BassFlute.__repr__()`

8.2.10 instrumenttools.BassSaxophone



class instrumenttools.**BassSaxophone**(***kwargs*)

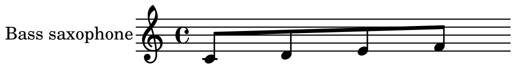
New in version 2.6. Abjad model of the bass saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.BassSaxophone()(staff)
BassSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Bass saxophone }
    \set Staff.shortInstrumentName = \markup { Bass sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



The bass saxophone is pitched in B-flat.

The bass saxophone targets staff context by default.

Read-only properties

BassSaxophone.**default_instrument_name**
Read-only default instrument name.

Return string.

BassSaxophone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

BassSaxophone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

BassSaxophone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

BassSaxophone.**is_primary_instrument**

BassSaxophone.**is_secondary_instrument**

BassSaxophone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

BassSaxophone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BassSaxophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BassSaxophone.**storage_format**
Storage format of Abjad object.

Return string.

BassSaxophone.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BassSaxophone.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

BassSaxophone.**all_clefs**

BassSaxophone.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BassSaxophone.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BassSaxophone.**pitch_range**

BassSaxophone.**primary_clefs**

BassSaxophone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BassSaxophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BassSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

BassSaxophone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BassSaxophone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BassSaxophone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

BassSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

BassSaxophone.**__call__**(**args*)

BassSaxophone.**__copy__**(**args*)

BassSaxophone.**__deepcopy__**(**args*)

BassSaxophone.**__delattr__**(**args*)

BassSaxophone.**__eq__**(*arg*)

BassSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BassSaxophone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BassSaxophone.**__hash__**()

BassSaxophone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

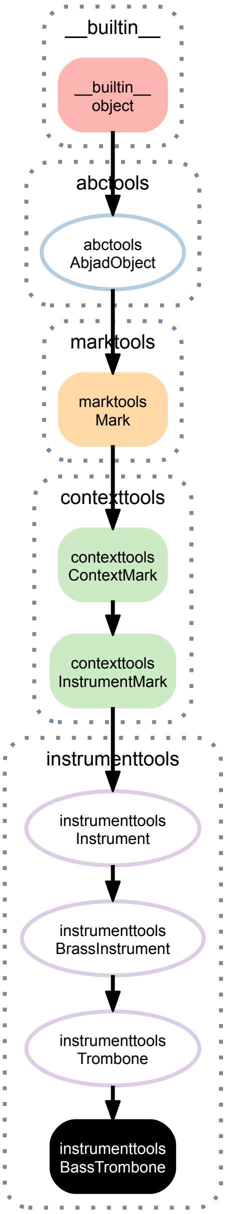
BassSaxophone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BassSaxophone.**__ne__**(*arg*)

BassSaxophone.**__repr__**()

8.2.11 instrumenttools.BassTrombone



class `instrumenttools.`**`BassTrombone`**`(`*`**kwargs`*`)`

New in version 2.0. Abjad model of the tenor trombone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.BassTrombone()(staff)
BassTrombone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Bass trombone }
    \set Staff.shortInstrumentName = \markup { Bass trb. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The tenor trombone targets staff context by default.

Read-only properties

`BassTrombone.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`BassTrombone.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`BassTrombone.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`BassTrombone.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`BassTrombone.`**`is_primary_instrument`**

`BassTrombone.`**`is_secondary_instrument`**

`BassTrombone.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`BassTrombone.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BassTrombone.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BassTrombone.**storage_format**

Storage format of Abjad object.

Return string.

BassTrombone.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BassTrombone.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

BassTrombone.**all_clefs**

BassTrombone.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BassTrombone.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BassTrombone.**pitch_range**

BassTrombone.**primary_clefs**

BassTrombone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BassTrombone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BassTrombone.**sounding_pitch_of_fingered_middle_c**

Methods

BassTrombone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

BassTrombone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BassTrombone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`BassTrombone.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`BassTrombone.`**`__call__`**`(*args)`

`BassTrombone.`**`__copy__`**`(*args)`

`BassTrombone.`**`__deepcopy__`**`(*args)`

`BassTrombone.`**`__delattr__`**`(*args)`

`BassTrombone.`**`__eq__`**`(arg)`

`BassTrombone.`**`__ge__`**`(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassTrombone.`**`__gt__`**`(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BassTrombone.`**`__hash__`**`()`

`BassTrombone.`**`__le__`**`(expr)`
Abjad objects by default do not implement this method.

Raise exception.

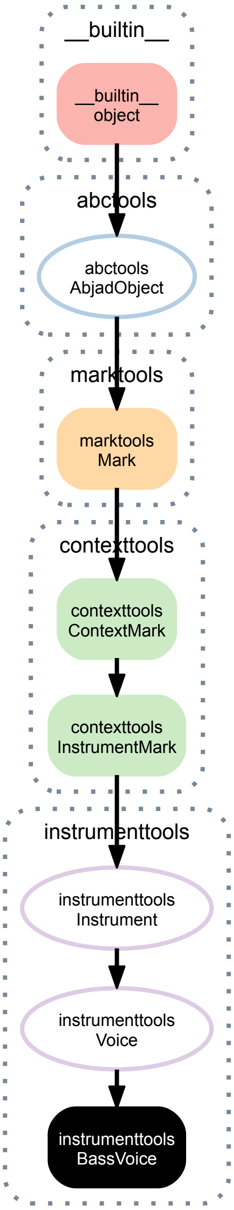
`BassTrombone.`**`__lt__`**`(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassTrombone.`**`__ne__`**`(arg)`

`BassTrombone.`**`__repr__`**`()`

8.2.12 instrumenttools.BassVoice



class `instrumenttools.BassVoice`(***kwargs*)

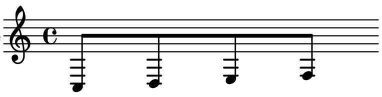
New in version 2.8. Abjad model of the bass voice:

```
>>> staff = Staff("c8 d8 e8 f8")
```

```
>>> instrumenttools.BassVoice()(staff)
BassVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Bass voice }
    \set Staff.shortInstrumentName = \markup { Bass }
    c8
    d8
    e8
    f8
}
```

```
>>> show(staff)
```



The bass voice targets staff context by default.

Read-only properties

BassVoice.**default_instrument_name**
Read-only default instrument name.

Return string.

BassVoice.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

BassVoice.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

BassVoice.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

BassVoice.**is_primary_instrument**

BassVoice.**is_secondary_instrument**

BassVoice.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

BassVoice.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

BassVoice.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BassVoice.**storage_format**
Storage format of Abjad object.

Return string.

BassVoice.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

BassVoice.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

BassVoice.**all_clefs**

BassVoice.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

BassVoice.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

BassVoice.**pitch_range**

BassVoice.**primary_clefs**

BassVoice.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

BassVoice.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

BassVoice.**sounding_pitch_of_fingered_middle_c**

Methods

BassVoice.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

BassVoice.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

BassVoice.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

BassVoice.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

BassVoice.**__call__**(**args*)

BassVoice.**__copy__**(**args*)

BassVoice.**__deepcopy__**(**args*)

BassVoice.**__delattr__**(**args*)

BassVoice.**__eq__**(*arg*)

BassVoice.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`BassVoice.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BassVoice.__hash__()`

`BassVoice.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

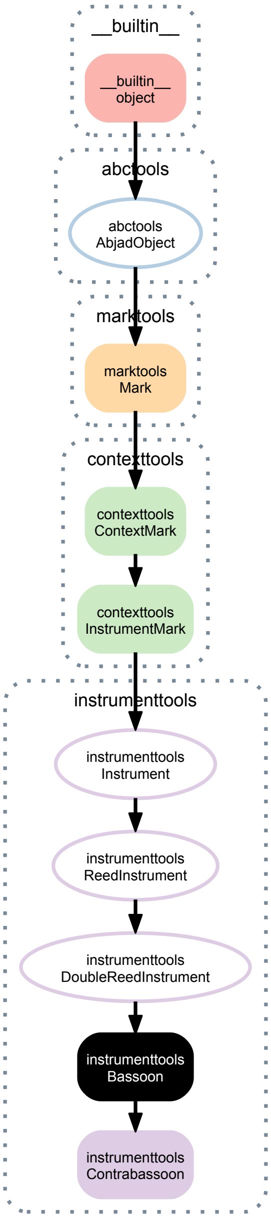
`BassVoice.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BassVoice.__ne__(arg)`

`BassVoice.__repr__()`

8.2.13 instrumenttools.Bassoon



class `instrumenttools.Bassoon(**kwargs)`

New in version 2.0. Abjad model of the bassoon:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Bassoon()(staff)
Bassoon()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Bassoon }
    \set Staff.shortInstrumentName = \markup { Bsn. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The bassoon targets staff context by default.

Read-only properties

`Bassoon.default_instrument_name`

Read-only default instrument name.

Return string.

`Bassoon.default_short_instrument_name`

Read-only default short instrument name.

Return string.

`Bassoon.effective_context`

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Bassoon.interval_of_transposition`

Read-only interval of transposition.

Return melodic diatonic interval.

`Bassoon.is_primary_instrument`

`Bassoon.is_secondary_instrument`

`Bassoon.is_transposing`

True when instrument is transposing. False otherwise.

Return boolean.

`Bassoon.lilypond_format`

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Bassoon.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Bassoon.**storage_format**
Storage format of Abjad object.

Return string.

Bassoon.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Bassoon.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Bassoon.**all_clefs**

Bassoon.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Bassoon.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Bassoon.**pitch_range**

Bassoon.**primary_clefs**

Bassoon.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Bassoon.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Bassoon.**sounding_pitch_of_fingered_middle_c**

Methods

Bassoon.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Bassoon.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Bassoon.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Bassoon.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Bassoon.**__call__**(**args*)

Bassoon.**__copy__**(**args*)

Bassoon.**__deepcopy__**(**args*)

Bassoon.**__delattr__**(**args*)

Bassoon.**__eq__**(*arg*)

Bassoon.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Bassoon.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Bassoon.**__hash__**()

Bassoon.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

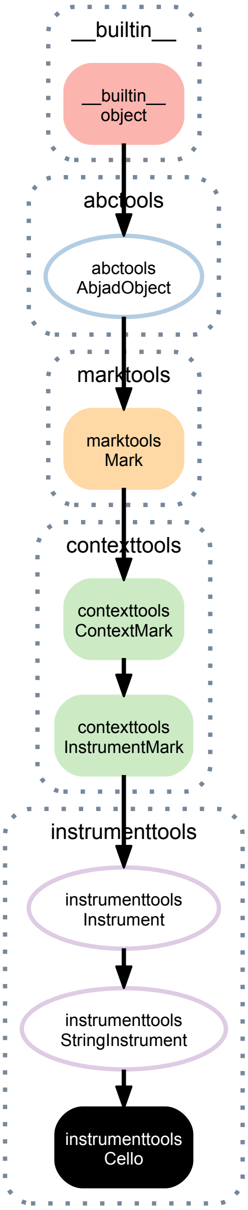
Bassoon.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Bassoon.**__ne__**(*arg*)

Bassoon.**__repr__**()

8.2.14 instrumenttools.Cello



class `instrumenttools.Cello(**kwargs)`

New in version 2.0. Abjad model of the cello:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Cello()(staff)
Cello()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Cello }
    \set Staff.shortInstrumentName = \markup { Vc. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The cello targets staff context by default.

Read-only properties

`Cello.default_instrument_name`
Read-only default instrument name.

Return string.

`Cello.default_short_instrument_name`
Read-only default short instrument name.

Return string.

`Cello.effective_context`
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Cello.interval_of_transposition`
Read-only interval of transposition.

Return melodic diatonic interval.

`Cello.is_primary_instrument`

`Cello.is_secondary_instrument`

`Cello.is_transposing`
True when instrument is transposing. False otherwise.

Return boolean.

`Cello.lilypond_format`
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Cello.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Cello.storage_format`
Storage format of Abjad object.

Return string.

`Cello.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Cello.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Cello.`**`all_clefs`**

`Cello.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Cello.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Cello.`**`pitch_range`**

`Cello.`**`primary_clefs`**

`Cello.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Cello.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Cello.**sounding_pitch_of_fingered_middle_c**

Methods

Cello.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Cello.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Cello.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Cello.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Cello.**__call__**(**args*)

Cello.**__copy__**(**args*)

Cello.**__deepcopy__**(**args*)

Cello.**__delattr__**(**args*)

Cello.**__eq__**(*arg*)

Cello.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Cello.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Cello.__hash__()`

`Cello.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

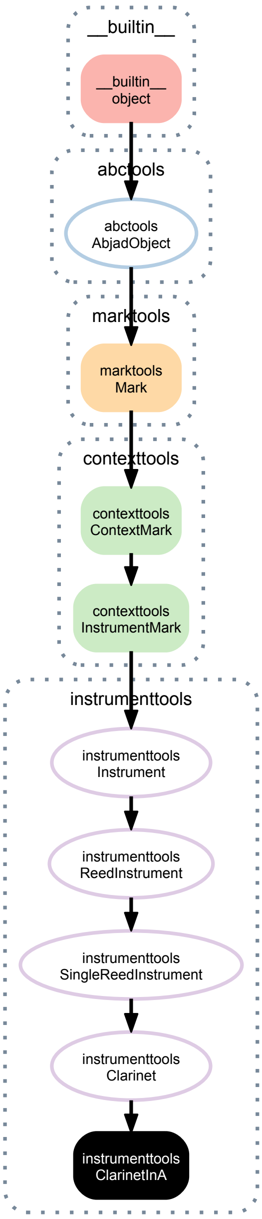
`Cello.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Cello.__ne__(arg)`

`Cello.__repr__()`

8.2.15 instrumenttools.ClarinetInA



class `instrumenttools.ClarinetInA(**kwargs)`

New in version 2.6. Abjad model of the clarinet in A:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ClarinetInA()(staff)
ClarinetInA()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in A }
    \set Staff.shortInstrumentName = \markup { Cl. A \natural }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The clarinet in A targets staff context by default.

Read-only properties

`ClarinetInA.default_instrument_name`

Read-only default instrument name.

Return string.

`ClarinetInA.default_short_instrument_name`

Read-only default short instrument name.

Return string.

`ClarinetInA.effective_context`

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`ClarinetInA.interval_of_transposition`

Read-only interval of transposition.

Return melodic diatonic interval.

`ClarinetInA.is_primary_instrument`

`ClarinetInA.is_secondary_instrument`

`ClarinetInA.is_transposing`

True when instrument is transposing. False otherwise.

Return boolean.

`ClarinetInA.lilypond_format`

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`ClarinetInA.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`ClarinetInA.storage_format`
Storage format of Abjad object.

Return string.

`ClarinetInA.target_context`
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`ClarinetInA.traditional_pitch_range`
Read-only traditional pitch range.

Return pitch range.

Read/write properties

`ClarinetInA.all_clefs`

`ClarinetInA.instrument_name`
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`ClarinetInA.instrument_name_markup`
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`ClarinetInA.pitch_range`

`ClarinetInA.`**`primary_clefs`**

`ClarinetInA.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`ClarinetInA.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`ClarinetInA.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`ClarinetInA.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`ClarinetInA.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`ClarinetInA.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`ClarinetInA.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

ClarinetInA.**__call__**(**args*)

ClarinetInA.**__copy__**(**args*)

ClarinetInA.**__deepcopy__**(**args*)

ClarinetInA.**__delattr__**(**args*)

ClarinetInA.**__eq__**(*arg*)

ClarinetInA.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ClarinetInA.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

ClarinetInA.**__hash__**()

ClarinetInA.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ClarinetInA.**__lt__**(*expr*)

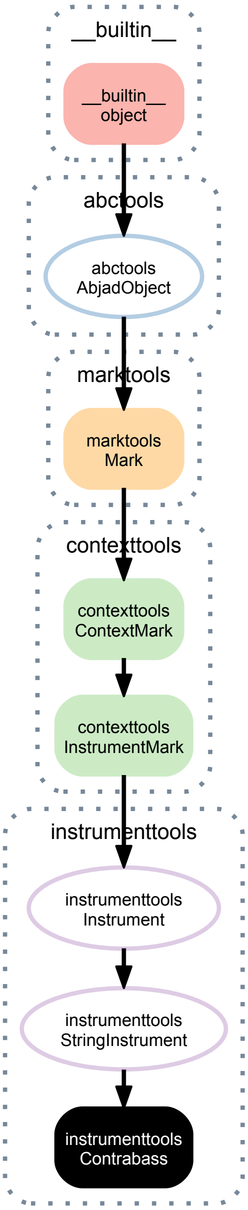
Abjad objects by default do not implement this method.

Raise exception.

ClarinetInA.**__ne__**(*arg*)

ClarinetInA.**__repr__**()

8.2.16 instrumenttools.Contrabass



class `instrumenttools.Contrabass`(***kwargs*)
New in version 2.0. Abjad model of the contrabass:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Contrabass()(staff)
Contrabass()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Contrabass }
    \set Staff.shortInstrumentName = \markup { Vb. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contrabass targets staff context by default.

Read-only properties

`Contrabass.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Contrabass.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Contrabass.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Contrabass.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Contrabass.`**`is_primary_instrument`**

`Contrabass.`**`is_secondary_instrument`**

`Contrabass.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Contrabass.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Contrabass.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Contrabass.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Contrabass.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Contrabass.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Contrabass.`**`all_clefs`**

`Contrabass.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Contrabass.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Contrabass.`**`pitch_range`**

`Contrabass.`**`primary_clefs`**

`Contrabass.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Contrabass.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Contrabass.**sounding_pitch_of_fingered_middle_c**

Methods

Contrabass.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Contrabass.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Contrabass.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Contrabass.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Contrabass.**__call__**(**args*)

Contrabass.**__copy__**(**args*)

Contrabass.**__deepcopy__**(**args*)

Contrabass.**__delattr__**(**args*)

Contrabass.**__eq__**(*arg*)

Contrabass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Contrabass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Contrabass.**__hash__**()

Contrabass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

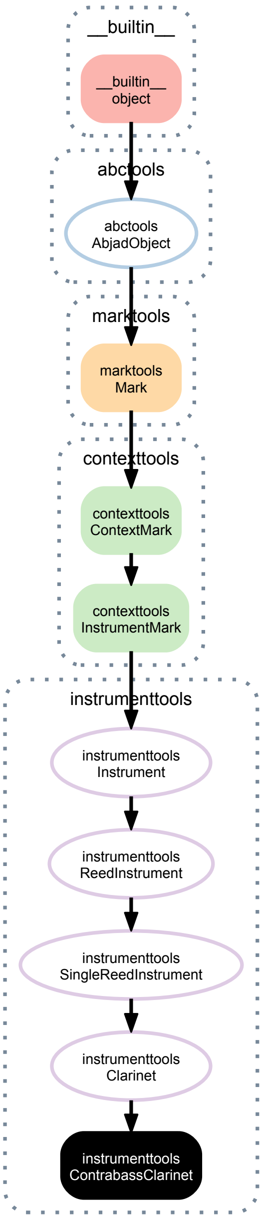
Contrabass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Contrabass.**__ne__**(*arg*)

Contrabass.**__repr__**()

8.2.17 instrumenttools.ContrabassClarinet



class instrumenttools.**ContrabassClarinet**(***kwargs*)

New in version 2.6. Abjad model of the contrassbass clarinet:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassClarinet()(staff)
ContrabassClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Contrabass clarinet }
    \set Staff.shortInstrumentName = \markup { Cbass cl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contrabass clarinet targets staff context by default.

Read-only properties

ContrabassClarinet.**default_instrument_name**

Read-only default instrument name.

Return string.

ContrabassClarinet.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

ContrabassClarinet.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

ContrabassClarinet.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

ContrabassClarinet.**is_primary_instrument**

ContrabassClarinet.**is_secondary_instrument**

ContrabassClarinet.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

ContrabassClarinet.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

ContrabassClarinet.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

ContrabassClarinet.**storage_format**
Storage format of Abjad object.

Return string.

ContrabassClarinet.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

ContrabassClarinet.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

ContrabassClarinet.**all_clefs**

ContrabassClarinet.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

ContrabassClarinet.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

ContrabassClarinet.**pitch_range**

ContrabassClarinet.**primary_clefs**

ContrabassClarinet.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

ContrabassClarinet.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

ContrabassClarinet.**sounding_pitch_of_fingered_middle_c**

Methods

ContrabassClarinet.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

ContrabassClarinet.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

ContrabassClarinet.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

ContrabassClarinet.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

ContrabassClarinet.**__call__**(*args*)

ContrabassClarinet.**__copy__**(*args*)

ContrabassClarinet.**__deepcopy__**(*args*)

ContrabassClarinet.**__delattr__**(*args*)

ContrabassClarinet.**__eq__**(*arg*)

ContrabassClarinet.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContrabassClarinet.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ContrabassClarinet.**__hash__**()

ContrabassClarinet.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

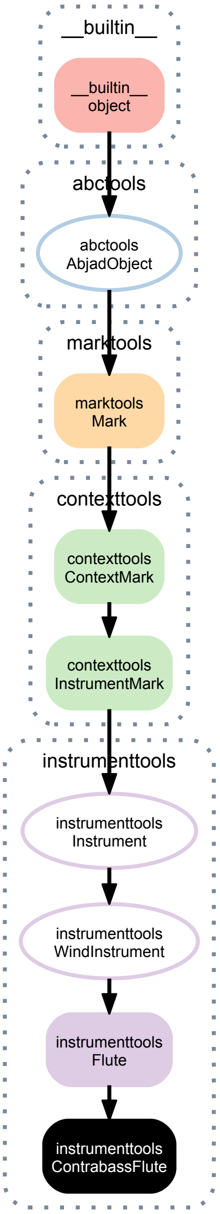
ContrabassClarinet.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContrabassClarinet.**__ne__**(*arg*)

ContrabassClarinet.**__repr__**()

8.2.18 instrumenttools.ContrabassFlute



class `instrumenttools.ContrabassFlute`(***kwargs*)

New in version 2.0. Abjad model of the contrabass flute:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassFlute()(staff)
ContrabassFlute()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Contrabass flute }
    \set Staff.shortInstrumentName = \markup { Cbass. fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contrabass flute targets staff context by default.

Read-only properties

ContrabassFlute.**default_instrument_name**
Read-only default instrument name.

Return string.

ContrabassFlute.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

ContrabassFlute.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

ContrabassFlute.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

ContrabassFlute.**is_primary_instrument**

ContrabassFlute.**is_secondary_instrument**

ContrabassFlute.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

ContrabassFlute.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

ContrabassFlute.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

ContrabassFlute.**storage_format**
Storage format of Abjad object.

Return string.

ContrabassFlute.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

ContrabassFlute.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

ContrabassFlute.**all_clefs**

ContrabassFlute.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

ContrabassFlute.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

ContrabassFlute.**pitch_range**

ContrabassFlute.**primary_clefs**

ContrabassFlute.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

ContrabassFlute.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

ContrabassFlute.**sounding_pitch_of_fingered_middle_c**

Methods

ContrabassFlute.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

ContrabassFlute.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

ContrabassFlute.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

ContrabassFlute.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

ContrabassFlute.**__call__**(**args*)

ContrabassFlute.**__copy__**(**args*)

ContrabassFlute.**__deepcopy__**(**args*)

ContrabassFlute.**__delattr__**(**args*)

ContrabassFlute.**__eq__**(*arg*)

ContrabassFlute.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ContrabassFlute.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ContrabassFlute.**__hash__**()

ContrabassFlute.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

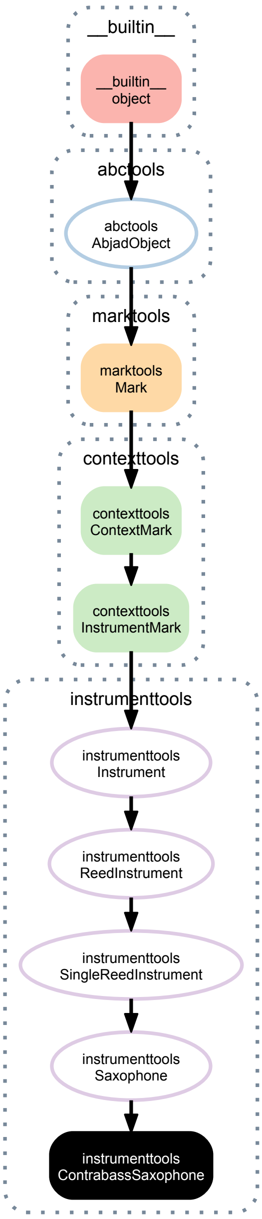
ContrabassFlute.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContrabassFlute.**__ne__**(*arg*)

ContrabassFlute.**__repr__**()

8.2.19 instrumenttools.ContrabassSaxophone



class `instrumenttools.`**`ContrabassSaxophone`**`(**kwargs)`

New in version 2.6. Abjad model of the bass saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContrabassSaxophone()(staff)
ContrabassSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Contrabass saxophone }
    \set Staff.shortInstrumentName = \markup { Cbass. sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contrabass saxophone is pitched in E-flat.

The contrabass saxophone targets staff context by default.

Read-only properties

`ContrabassSaxophone.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`ContrabassSaxophone.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`ContrabassSaxophone.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`ContrabassSaxophone.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`ContrabassSaxophone.`**`is_primary_instrument`**

`ContrabassSaxophone.`**`is_secondary_instrument`**

`ContrabassSaxophone.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`ContrabassSaxophone.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`ContrabassSaxophone.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`ContrabassSaxophone.`**`storage_format`**
Storage format of Abjad object.

Return string.

`ContrabassSaxophone.`**`target_context`**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`ContrabassSaxophone.`**`traditional_pitch_range`**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

`ContrabassSaxophone.`**`all_clefs`**

`ContrabassSaxophone.`**`instrument_name`**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`ContrabassSaxophone.`**`instrument_name_markup`**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

ContrabassSaxophone.**pitch_range**

ContrabassSaxophone.**primary_clefs**

ContrabassSaxophone.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

ContrabassSaxophone.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

ContrabassSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

ContrabassSaxophone.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

ContrabassSaxophone.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

ContrabassSaxophone.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

ContrabassSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

ContrabassSaxophone.**__call__**(**args*)

ContrabassSaxophone.**__copy__**(**args*)

ContrabassSaxophone.**__deepcopy__**(**args*)

ContrabassSaxophone.**__delattr__**(**args*)

ContrabassSaxophone.**__eq__**(*arg*)

ContrabassSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContrabassSaxophone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ContrabassSaxophone.**__hash__**()

ContrabassSaxophone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

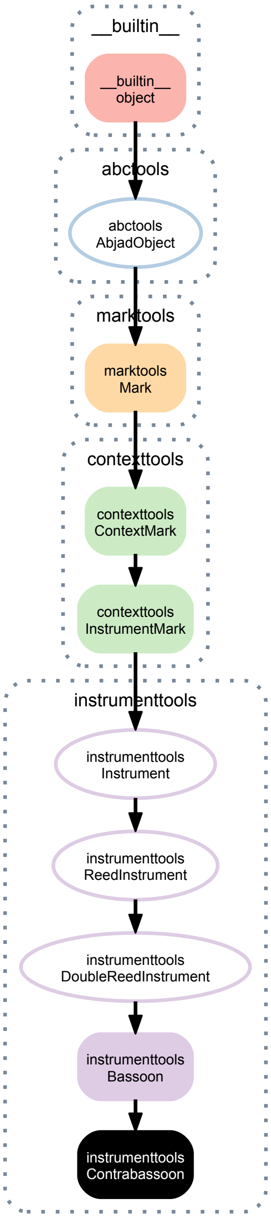
ContrabassSaxophone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ContrabassSaxophone.**__ne__**(*arg*)

ContrabassSaxophone.**__repr__**()

8.2.20 instrumenttools.Contrabassoon



class `instrumenttools.Contrabassoon(**kwargs)`

New in version 2.0. Abjad model of the contrabassoon:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Contrabassoon()(staff)
Contrabassoon()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Contrabassoon }
    \set Staff.shortInstrumentName = \markup { Contrabsn. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contrabassoon targets staff context by default.

Read-only properties

`Contrabassoon.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Contrabassoon.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Contrabassoon.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Contrabassoon.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Contrabassoon.`**`is_primary_instrument`**

`Contrabassoon.`**`is_secondary_instrument`**

`Contrabassoon.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Contrabassoon.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Contrabassoon.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Contrabassoon.`**`storage_format`**
Storage format of Abjad object.

Return string.

Contrabassoon.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Contrabassoon.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Contrabassoon.**all_clefs**

Contrabassoon.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Contrabassoon.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Contrabassoon.**pitch_range**

Contrabassoon.**primary_clefs**

Contrabassoon.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Contrabassoon.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Contrabassoon.**sounding_pitch_of_fingered_middle_c**

Methods

Contrabassoon.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Contrabassoon.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Contrabassoon.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Contrabassoon.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Contrabassoon.**__call__**(**args*)

Contrabassoon.**__copy__**(**args*)

Contrabassoon.**__deepcopy__**(**args*)

Contrabassoon.**__delattr__**(**args*)

Contrabassoon.**__eq__**(*arg*)

Contrabassoon.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Contrabassoon.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Contrabassoon.__hash__()`

`Contrabassoon.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

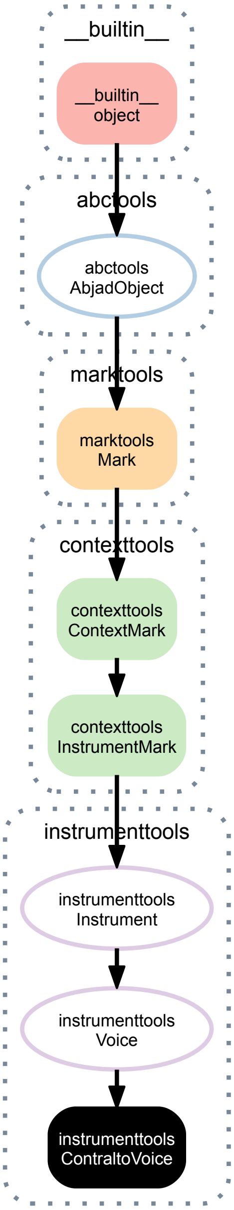
`Contrabassoon.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Contrabassoon.__ne__(arg)`

`Contrabassoon.__repr__()`

8.2.21 instrumenttools.ContraltoVoice



class instrumenttools.**ContraltoVoice**(***kwargs*)
New in version 2.8. Abjad model of the contralto voice:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.ContraltoVoice()(staff)
ContraltoVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Contralto voice }
    \set Staff.shortInstrumentName = \markup { Contralto }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The contralto voice targets staff context by default.

Read-only properties

ContraltoVoice.**default_instrument_name**
Read-only default instrument name.

Return string.

ContraltoVoice.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

ContraltoVoice.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

ContraltoVoice.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

ContraltoVoice.**is_primary_instrument**

ContraltoVoice.**is_secondary_instrument**

ContraltoVoice.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

ContraltoVoice.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

ContraltoVoice.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

ContraltoVoice.**storage_format**
Storage format of Abjad object.

Return string.

ContraltoVoice.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

ContraltoVoice.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

ContraltoVoice.**all_clefs**

ContraltoVoice.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

ContraltoVoice.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

ContraltoVoice.**pitch_range**

ContraltoVoice.**primary_clefs**

ContraltoVoice.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

ContraltoVoice.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

ContraltoVoice.**sounding_pitch_of_fingered_middle_c**

Methods

ContraltoVoice.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

ContraltoVoice.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

ContraltoVoice.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

ContraltoVoice.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

`ContraltoVoice.__call__(*args)`

`ContraltoVoice.__copy__(*args)`

`ContraltoVoice.__deepcopy__(*args)`

`ContraltoVoice.__delattr__(*args)`

`ContraltoVoice.__eq__(arg)`

`ContraltoVoice.__ge__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`ContraltoVoice.__gt__(expr)`

Abjad objects by default do not implement this method.

Raise exception

`ContraltoVoice.__hash__()`

`ContraltoVoice.__le__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`ContraltoVoice.__lt__(expr)`

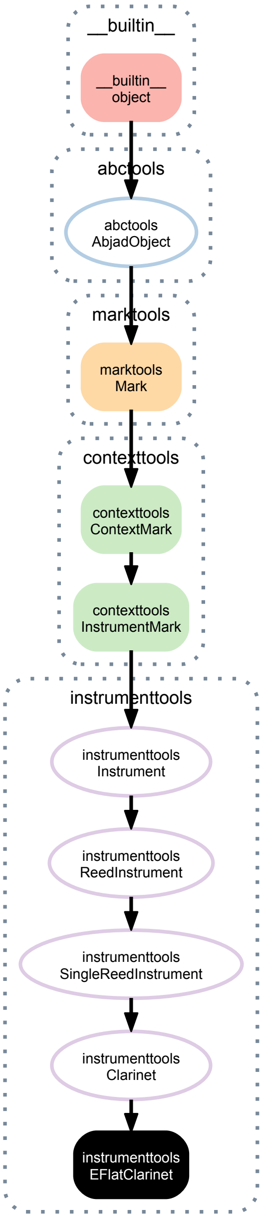
Abjad objects by default do not implement this method.

Raise exception.

`ContraltoVoice.__ne__(arg)`

`ContraltoVoice.__repr__()`

8.2.22 instrumenttools.EFlatClarinet



class `instrumenttools.`**`EFlatClarinet`**(***kwargs*)

New in version 2.0. Abjad model of the E-flat clarinet:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.EFlatClarinet()(staff)
EFlatClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in E-flat }
    \set Staff.shortInstrumentName = \markup { Cl. E-flat }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The E-flat clarinet targets staff context by default.

Read-only properties

EFlatClarinet.**default_instrument_name**

Read-only default instrument name.

Return string.

EFlatClarinet.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

EFlatClarinet.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

EFlatClarinet.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

EFlatClarinet.**is_primary_instrument**

EFlatClarinet.**is_secondary_instrument**

EFlatClarinet.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

EFlatClarinet.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

EFlatClarinet.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

EFlatClarinet.**storage_format**

Storage format of Abjad object.

Return string.

EFlatClarinet.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

EFlatClarinet.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

EFlatClarinet.**all_clefs**

EFlatClarinet.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

EFlatClarinet.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

EFlatClarinet.**pitch_range**

EFlatClarinet.**primary_clefs**

EFlatClarinet.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

EFlatClarinet.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

EFlatClarinet.**sounding_pitch_of_fingered_middle_c**

Methods

EFlatClarinet.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

EFlatClarinet.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

EFlatClarinet.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

EFlatClarinet.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

EFlatClarinet.**__call__**(**args*)

EFlatClarinet.**__copy__**(**args*)

EFlatClarinet.**__deepcopy__**(**args*)

EFlatClarinet.**__delattr__**(**args*)

EFlatClarinet.**__eq__**(*arg*)

EFlatClarinet.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

EFlatClarinet.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

EFlatClarinet.**__hash__**()

EFlatClarinet.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

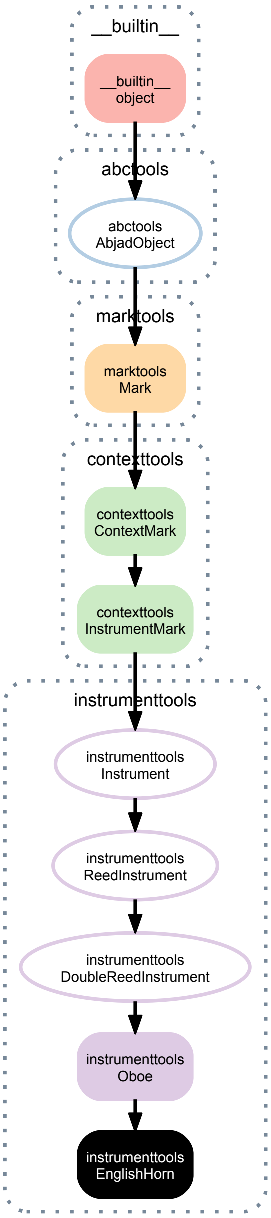
EFlatClarinet.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

EFlatClarinet.**__ne__**(*arg*)

EFlatClarinet.**__repr__**()

8.2.23 instrumenttools.EnglishHorn



class `instrumenttools.`**`EnglishHorn`**`(`***kwargs*`)`
New in version 2.0. Abjad model of the English horn:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.EnglishHorn()(staff)
EnglishHorn()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { English horn }
    \set Staff.shortInstrumentName = \markup { Eng. hn. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The English horn targets staff context by default.

Read-only properties

`EnglishHorn.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`EnglishHorn.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`EnglishHorn.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`EnglishHorn.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`EnglishHorn.`**`is_primary_instrument`**

`EnglishHorn.`**`is_secondary_instrument`**

`EnglishHorn.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`EnglishHorn.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

EnglishHorn.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

EnglishHorn.**storage_format**
Storage format of Abjad object.

Return string.

EnglishHorn.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

EnglishHorn.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

EnglishHorn.**all_clefs**

EnglishHorn.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

EnglishHorn.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

EnglishHorn.**pitch_range**

`EnglishHorn.`**`primary_clefs`**

`EnglishHorn.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`EnglishHorn.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`EnglishHorn.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`EnglishHorn.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`EnglishHorn.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`EnglishHorn.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`EnglishHorn.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

EnglishHorn.**__call__**(*args)

EnglishHorn.**__copy__**(*args)

EnglishHorn.**__deepcopy__**(*args)

EnglishHorn.**__delattr__**(*args)

EnglishHorn.**__eq__**(arg)

EnglishHorn.**__ge__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

EnglishHorn.**__gt__**(expr)
Abjad objects by default do not implement this method.

Raise exception

EnglishHorn.**__hash__**()

EnglishHorn.**__le__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

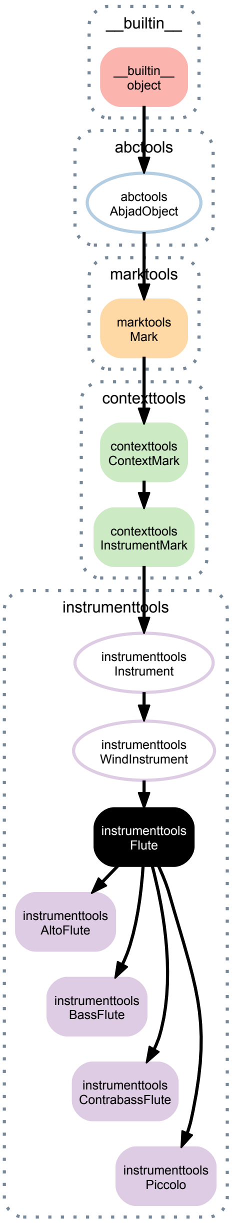
EnglishHorn.**__lt__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

EnglishHorn.**__ne__**(arg)

EnglishHorn.**__repr__**()

8.2.24 instrumenttools.Flute



class instrumenttools.**Flute**(***kwargs*)

New in version 2.0. Abjad model of the flute:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Flute()(staff)
Flute()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Flute }
    \set Staff.shortInstrumentName = \markup { Fl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The flute targets staff context by default.

Read-only properties

`Flute.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Flute.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Flute.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Flute.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Flute.`**`is_primary_instrument`**

`Flute.`**`is_secondary_instrument`**

`Flute.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Flute.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Flute.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Flute.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Flute.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Flute.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Flute.`**`all_clefs`**

`Flute.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Flute.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Flute.`**`pitch_range`**

`Flute.`**`primary_clefs`**

`Flute.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Flute.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Flute.**sounding_pitch_of_fingered_middle_c**

Methods

Flute.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Flute.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Flute.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Flute.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Flute.**__call__**(**args*)

Flute.**__copy__**(**args*)

Flute.**__deepcopy__**(**args*)

Flute.**__delattr__**(**args*)

Flute.**__eq__**(*arg*)

Flute.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Flute.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Flute.__hash__()`

`Flute.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

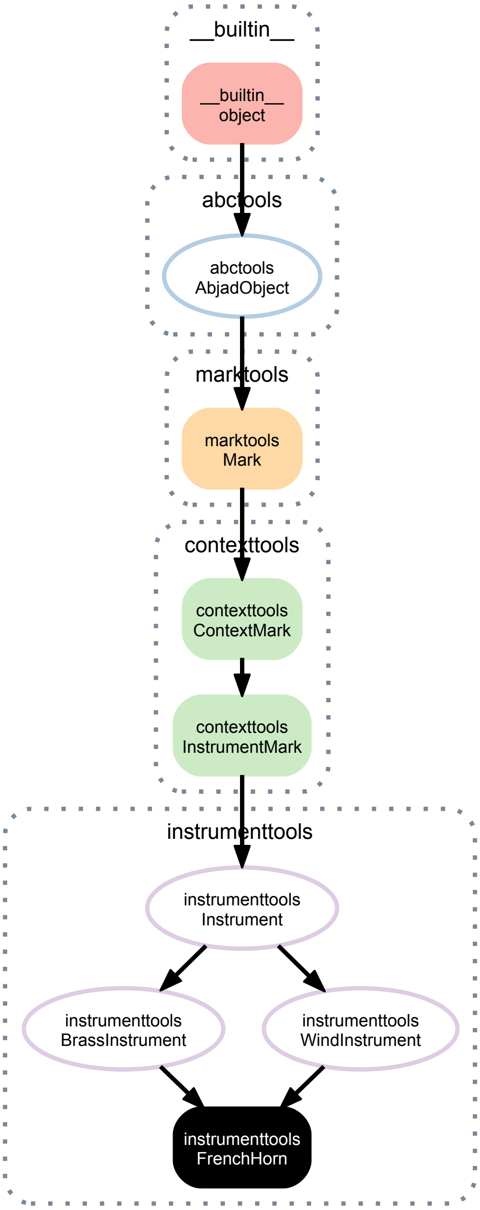
`Flute.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Flute.__ne__(arg)`

`Flute.__repr__()`

8.2.25 instrumenttools.FrenchHorn



class `instrumenttools.FrenchHorn(**kwargs)`

New in version 2.0. Abjad model of the French horn:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.FrenchHorn()(staff)
FrenchHorn()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Horn }
    \set Staff.shortInstrumentName = \markup { Hn. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The French horn targets staff context by default.

Read-only properties

`FrenchHorn.default_instrument_name`

Read-only default instrument name.

Return string.

`FrenchHorn.default_short_instrument_name`

Read-only default short instrument name.

Return string.

`FrenchHorn.effective_context`

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`FrenchHorn.interval_of_transposition`

Read-only interval of transposition.

Return melodic diatonic interval.

`FrenchHorn.is_primary_instrument`

`FrenchHorn.is_secondary_instrument`

`FrenchHorn.is_transposing`

True when instrument is transposing. False otherwise.

Return boolean.

`FrenchHorn.lilypond_format`

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

FrenchHorn.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

FrenchHorn.**storage_format**

Storage format of Abjad object.

Return string.

FrenchHorn.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

FrenchHorn.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

FrenchHorn.**all_clefs**

FrenchHorn.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

FrenchHorn.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

FrenchHorn.**pitch_range**

FrenchHorn.**primary_clefs**

FrenchHorn.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

FrenchHorn.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

FrenchHorn.**sounding_pitch_of_fingered_middle_c**

Methods

FrenchHorn.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

FrenchHorn.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

FrenchHorn.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

FrenchHorn.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

FrenchHorn.**__call__**(*args)

FrenchHorn.**__copy__**(*args)

FrenchHorn.**__deepcopy__**(*args)

FrenchHorn.**__delattr__**(*args)

FrenchHorn.**__eq__**(arg)

FrenchHorn.**__ge__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

FrenchHorn.**__gt__**(expr)
Abjad objects by default do not implement this method.

Raise exception

FrenchHorn.**__hash__**()

FrenchHorn.**__le__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

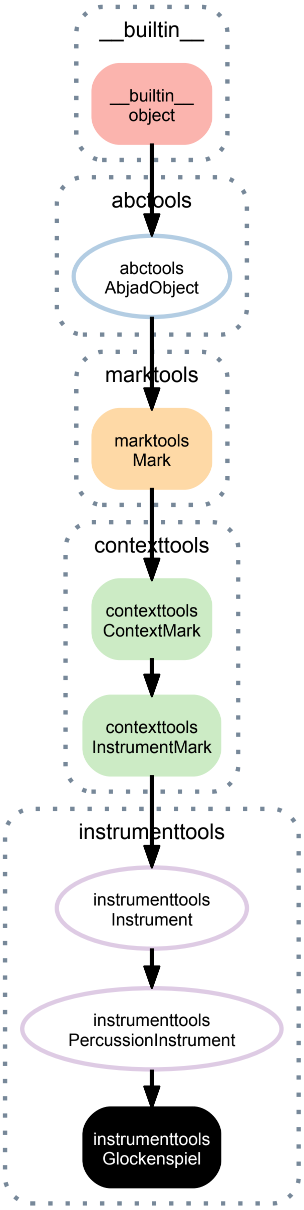
FrenchHorn.**__lt__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

FrenchHorn.**__ne__**(arg)

FrenchHorn.**__repr__**()

8.2.26 instrumenttools.Glockenspiel



class `instrumenttools.Glockenspiel` (***kwargs*)

New in version 2.0. Abjad model of the glockenspiel:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Glockenspiel()(staff)
Glockenspiel()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Glockenspiel }
    \set Staff.shortInstrumentName = \markup { Gkspl. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The glockenspiel targets staff context by default.

Read-only properties

Glockenspiel.**default_instrument_name**
Read-only default instrument name.

Return string.

Glockenspiel.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Glockenspiel.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Glockenspiel.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Glockenspiel.**is_primary_instrument**

Glockenspiel.**is_secondary_instrument**

Glockenspiel.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Glockenspiel.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Glockenspiel.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Glockenspiel.**storage_format**
Storage format of Abjad object.

Return string.

Glockenspiel.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Glockenspiel.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Glockenspiel.**all_clefs**

Glockenspiel.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Glockenspiel.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Glockenspiel.**pitch_range**

Glockenspiel.**primary_clefs**

Glockenspiel.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Glockenspiel.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Glockenspiel.**sounding_pitch_of_fingered_middle_c**

Methods

Glockenspiel.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

Glockenspiel.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Glockenspiel.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Glockenspiel.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

Glockenspiel.**__call__**(**args*)

Glockenspiel.**__copy__**(**args*)

Glockenspiel.**__deepcopy__**(**args*)

Glockenspiel.**__delattr__**(**args*)

Glockenspiel.**__eq__**(*arg*)

Glockenspiel.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Glockenspiel.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Glockenspiel.**__hash__**()

Glockenspiel.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

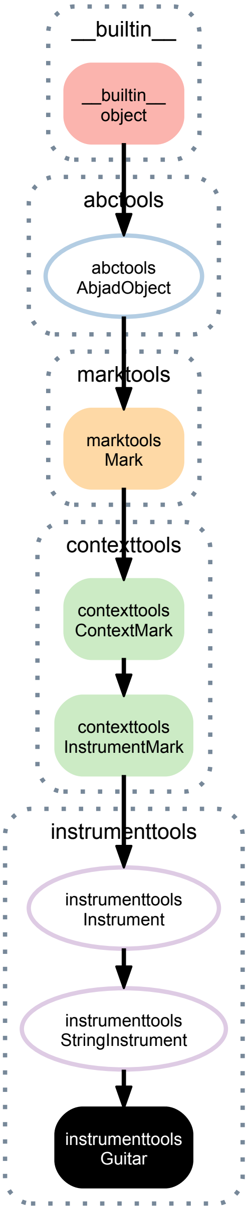
Glockenspiel.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Glockenspiel.**__ne__**(*arg*)

Glockenspiel.**__repr__**()

8.2.27 instrumenttools.Guitar



class `instrumenttools.`**`Guitar`**`(`***kwargs*`)`

New in version 2.0. Abjad model of the guitar:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Guitar()(staff)
Guitar()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Guitar }
    \set Staff.shortInstrumentName = \markup { Gt. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The guitar targets staff context by default.

Read-only properties

`Guitar.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`Guitar.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`Guitar.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Guitar.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`Guitar.`**`is_primary_instrument`**

`Guitar.`**`is_secondary_instrument`**

`Guitar.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`Guitar.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Guitar.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Guitar.**storage_format**

Storage format of Abjad object.

Return string.

Guitar.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Guitar.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Guitar.**all_clefs**

Guitar.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Guitar.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Guitar.**pitch_range**

`Guitar.`**`primary_clefs`**

`Guitar.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Guitar.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Guitar.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Guitar.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`Guitar.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Guitar.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Guitar.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`Guitar.__call__(*args)`

`Guitar.__copy__(*args)`

`Guitar.__deepcopy__(*args)`

`Guitar.__delattr__(*args)`

`Guitar.__eq__(arg)`

`Guitar.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Guitar.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Guitar.__hash__()`

`Guitar.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

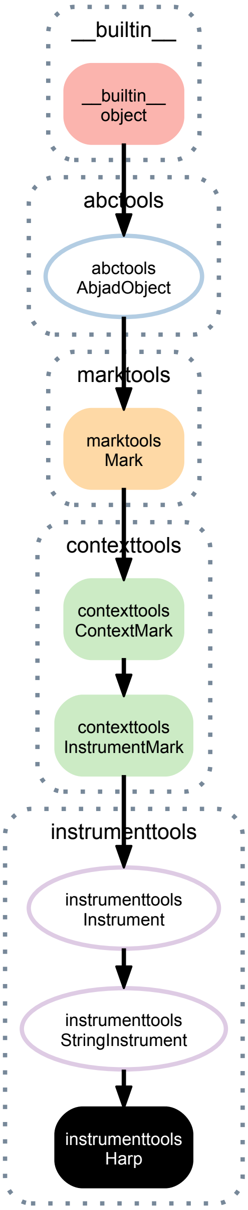
`Guitar.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Guitar.__ne__(arg)`

`Guitar.__repr__()`

8.2.28 instrumenttools.Harp



class `instrumenttools.Harp`(*target_context=None*, ***kwargs*)

New in version 2.0. Abjad model of the harp:

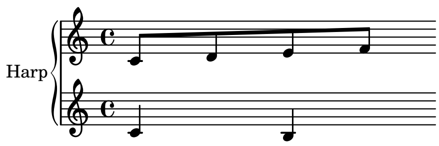
```
>>> piano_staff = scoretools.PianoStaff([Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Harp()(piano_staff)
Harp()(PianoStaff<<2>>)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \set PianoStaff.instrumentName = \markup { Harp }
    \set PianoStaff.shortInstrumentName = \markup { Hp. }
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        b4
```

```
    }
>>
```

```
>>> show(piano_staff)
```



The harp targets piano staff context by default.

Read-only properties

Harp.default_instrument_name
Read-only default instrument name.

Return string.

Harp.default_short_instrument_name
Read-only default short instrument name.

Return string.

Harp.effective_context
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Harp.interval_of_transposition
Read-only interval of transposition.

Return melodic diatonic interval.

Harp.is_primary_instrument

Harp.is_secondary_instrument

Harp.is_transposing
True when instrument is transposing. False otherwise.

Return boolean.

Harp.lilypond_format
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Harp.start_component
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Harp.**storage_format**
Storage format of Abjad object.

Return string.

Harp.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Harp.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Harp.**all_clefs**

Harp.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Harp.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Harp.**pitch_range**

Harp.**primary_clefs**

Harp.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Harp.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Harp.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Harp.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`Harp.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Harp.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Harp.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`Harp.`**`__call__`**`(`**args*`)`

`Harp.`**`__copy__`**`(`**args*`)`

`Harp.`**`__deepcopy__`**`(`**args*`)`

`Harp.`**`__delattr__`**`(`**args*`)`

`Harp.`**`__eq__`**`(`*arg*`)`

Harp.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Harp.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Harp.**__hash__**()

Harp.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

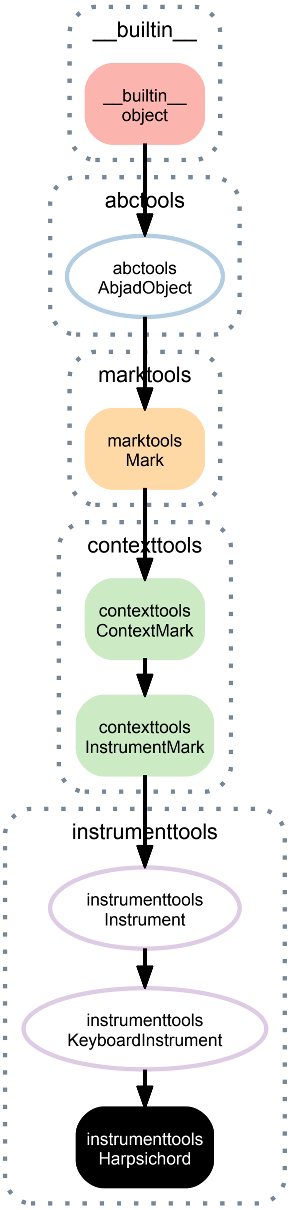
Harp.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Harp.**__ne__**(*arg*)

Harp.**__repr__**()

8.2.29 instrumenttools.Harpsichord



class `instrumenttools.Harpsichord`(*target_context=None*, ***kwargs*)

New in version 2.5. Abjad model of the harpsichord:

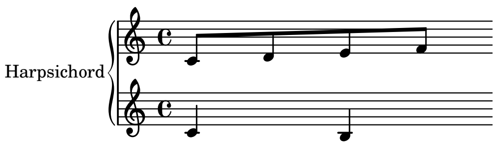
```
>>> piano_staff = scoretools.PianoStaff([Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Harpsichord()(piano_staff)
Harpsichord()(PianoStaff<<2>>)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \set PianoStaff.instrumentName = \markup { Harpsichord }
    \set PianoStaff.shortInstrumentName = \markup { Hpschd. }
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        b4
```

```
    }
>>
```

```
>>> show(piano_staff)
```



The harpsichord targets piano staff context by default.

Return instrument.

Read-only properties

Harpsichord.**default_instrument_name**

Read-only default instrument name.

Return string.

Harpsichord.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

Harpsichord.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c’4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Harpsichord.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

Harpsichord.**is_primary_instrument**

Harpsichord.**is_secondary_instrument**

Harpsichord.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

Harpsichord.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark(’Flute’, ’Fl.’)
>>> instrument.lilypond_format
[’\set Staff.instrumentName = \markup { Flute }’,
    ’\set Staff.shortInstrumentName = \markup { Fl. }’]
```

Return list.

Harpsichord.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c’4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Harpsichord.**storage_format**
Storage format of Abjad object.

Return string.

Harpsichord.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Harpsichord.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Harpsichord.**all_clefs**

Harpsichord.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Harpsichord.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Harpsichord.**pitch_range**

Harpsichord.**primary_clefs**

Harpsichord.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Harpsichord.`**`short_instrument_name_markup`**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Harpsichord.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Harpsichord.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`Harpsichord.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Harpsichord.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Harpsichord.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`Harpsichord.`**`__call__`**`(`**args*`)`

`Harpsichord.`**`__copy__`**`(`**args*`)`

`Harpsichord.`**`__deepcopy__`**`(`**args*`)`

Harpsichord.**__delattr__**(**args*)

Harpsichord.**__eq__**(*arg*)

Harpsichord.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Harpsichord.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Harpsichord.**__hash__**()

Harpsichord.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

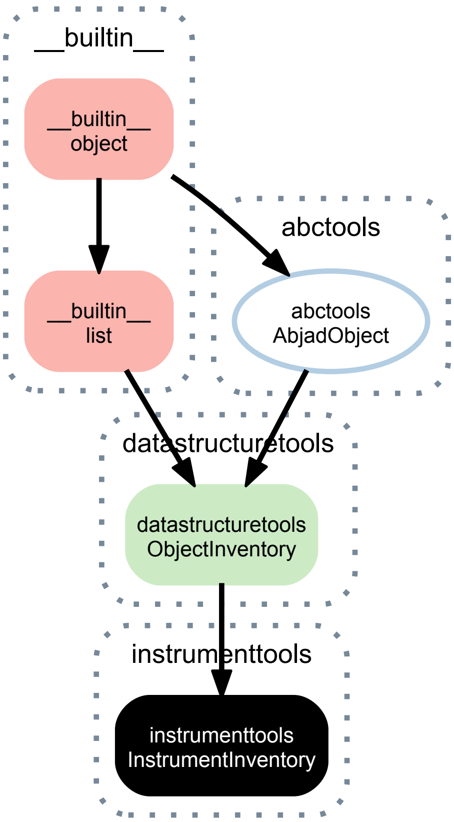
Harpsichord.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Harpsichord.**__ne__**(*arg*)

Harpsichord.**__repr__**()

8.2.30 instrumenttools.InstrumentInventory



class instrumenttools.**InstrumentInventory**(*tokens=None*, *name=None*)
New in version 2.8. Abjad model of an ordered list of instruments:

```
>>> inventory = instrumenttools.InstrumentInventory(
...     [instrumenttools.Flute(), instrumenttools.Guitar()])
```

```
>>> inventory
InstrumentInventory([Flute(), Guitar()])
```

Instrument inventories implement list interface and are mutable.

Read-only properties

`InstrumentInventory.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`InstrumentInventory.`**`name`**
Read / write name of inventory.

Methods

`InstrumentInventory.`**`append`**`(`*token*`)`
Change *token* to item and append.

`InstrumentInventory.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`InstrumentInventory.`**`extend`**`(`*tokens*`)`
Change *tokens* to items and extend.

`InstrumentInventory.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`InstrumentInventory.`**`insert`**`()`
L.insert(index, object) – insert object before index

`InstrumentInventory.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`InstrumentInventory.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`InstrumentInventory.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`InstrumentInventory.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`InstrumentInventory.`**`__add__`**`()`
x.__add__(y) <==> x+y

`InstrumentInventory.`**`__contains__`**`(`*token*`)`

`InstrumentInventory.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`InstrumentInventory.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`InstrumentInventory.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

InstrumentInventory.**__ge__**()
x.__ge__(y) <==> x>=y

InstrumentInventory.**__getitem__**()
x.__getitem__(y) <==> x[y]

InstrumentInventory.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

InstrumentInventory.**__gt__**()
x.__gt__(y) <==> x>y

InstrumentInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

InstrumentInventory.**__imul__**()
x.__imul__(y) <==> x*=y

InstrumentInventory.**__iter__**() <==> *iter(x)*

InstrumentInventory.**__le__**()
x.__le__(y) <==> x<=y

InstrumentInventory.**__len__**() <==> *len(x)*

InstrumentInventory.**__lt__**()
x.__lt__(y) <==> x<y

InstrumentInventory.**__mul__**()
x.__mul__(n) <==> x*n

InstrumentInventory.**__ne__**()
x.__ne__(y) <==> x!=y

InstrumentInventory.**__repr__**()

InstrumentInventory.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

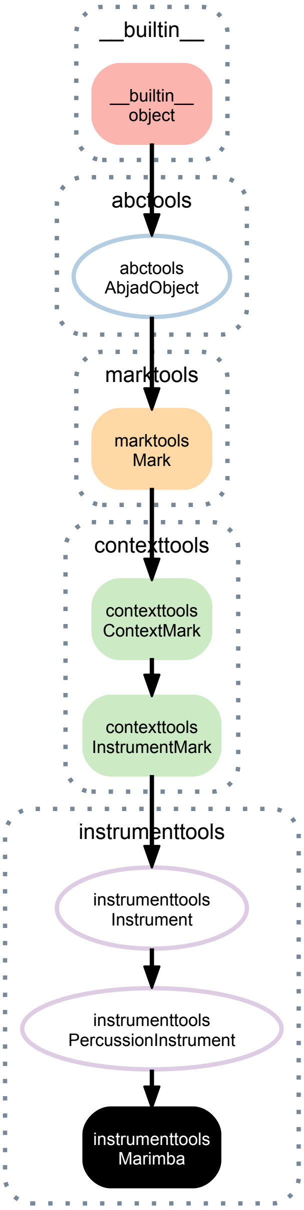
InstrumentInventory.**__rmul__**()
x.__rmul__(n) <==> n*x

InstrumentInventory.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

InstrumentInventory.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

8.2.31 instrumenttools.Marimba



class `instrumenttools.Marimba`(***kwargs*)

New in version 2.0. Abjad model of the marimba:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Marimba()(staff)
Marimba()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Marimba }
    \set Staff.shortInstrumentName = \markup { Mb. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The marimba targets staff context by default.

Read-only properties

`Marimba.default_instrument_name`
Read-only default instrument name.

Return string.

`Marimba.default_short_instrument_name`
Read-only default short instrument name.

Return string.

`Marimba.effective_context`
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Marimba.interval_of_transposition`
Read-only interval of transposition.

Return melodic diatonic interval.

`Marimba.is_primary_instrument`

`Marimba.is_secondary_instrument`

`Marimba.is_transposing`
True when instrument is transposing. False otherwise.

Return boolean.

`Marimba.lilypond_format`
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Marimba.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Marimba.storage_format`
Storage format of Abjad object.

Return string.

Marimba.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Marimba.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Marimba.**all_clefs**

Marimba.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Marimba.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Marimba.**pitch_range**

Marimba.**primary_clefs**

Marimba.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Marimba.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Marimba.**sounding_pitch_of_fingered_middle_c**

Methods

Marimba.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

Marimba.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Marimba.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Marimba.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

Marimba.**__call__**(**args*)

Marimba.**__copy__**(**args*)

Marimba.**__deepcopy__**(**args*)

Marimba.**__delattr__**(**args*)

Marimba.**__eq__**(*arg*)

Marimba.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`Marimba.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Marimba.__hash__()`

`Marimba.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

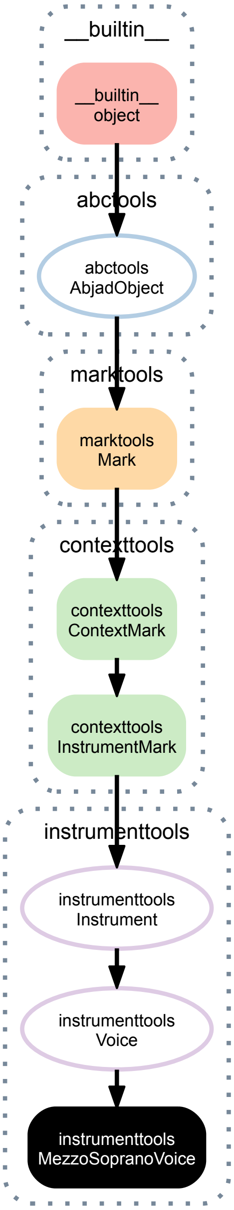
`Marimba.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Marimba.__ne__(arg)`

`Marimba.__repr__()`

8.2.32 instrumenttools.MezzoSopranoVoice



class instrumenttools.**MezzoSopranoVoice**(***kwargs*)

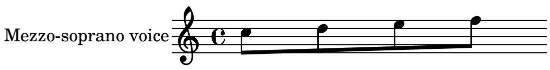
New in version 2.8. Abjad model of the mezzo-soprano voice:

```
>>> staff = Staff("c''8 d''8 e''8 f''8")
```

```
>>> instrumenttools.MezzoSopranoVoice()(staff)
MezzoSopranoVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Mezzo-soprano voice }
    \set Staff.shortInstrumentName = \markup { Mezzo-soprano }
    c''8
    d''8
    e''8
    f''8
}
```

```
>>> show(staff)
```



The mezzo-soprano voice targets staff context by default.

Read-only properties

MezzoSopranoVoice.**default_instrument_name**

Read-only default instrument name.

Return string.

MezzoSopranoVoice.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

MezzoSopranoVoice.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

MezzoSopranoVoice.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

MezzoSopranoVoice.**is_primary_instrument**

MezzoSopranoVoice.**is_secondary_instrument**

MezzoSopranoVoice.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

MezzoSopranoVoice.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

MezzoSopranoVoice.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

MezzoSopranoVoice.**storage_format**

Storage format of Abjad object.

Return string.

MezzoSopranoVoice.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

MezzoSopranoVoice.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

MezzoSopranoVoice.**all_clefs**

MezzoSopranoVoice.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

MezzoSopranoVoice.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

MezzoSopranoVoice.**pitch_range**

MezzoSopranoVoice.**primary_clefs**

MezzoSopranoVoice.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

MezzoSopranoVoice.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

MezzoSopranoVoice.**sounding_pitch_of_fingered_middle_c**

Methods

MezzoSopranoVoice.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

MezzoSopranoVoice.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

MezzoSopranoVoice.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

MezzoSopranoVoice.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

MezzoSopranoVoice.**__call__**(**args*)

MezzoSopranoVoice.**__copy__**(**args*)

MezzoSopranoVoice.**__deepcopy__**(**args*)

MezzoSopranoVoice.**__delattr__**(**args*)

MezzoSopranoVoice.**__eq__**(*arg*)

MezzoSopranoVoice.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MezzoSopranoVoice.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MezzoSopranoVoice.**__hash__**()

MezzoSopranoVoice.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

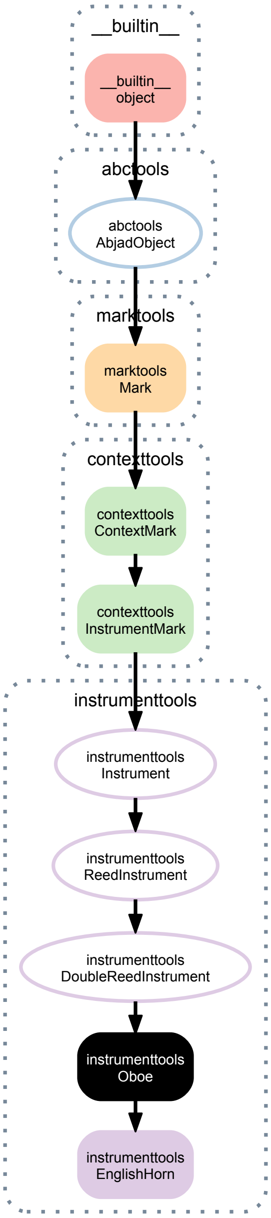
MezzoSopranoVoice.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MezzoSopranoVoice.**__ne__**(*arg*)

MezzoSopranoVoice.**__repr__**()

8.2.33 instrumenttools.Oboe



class `instrumenttools.Oboe` (***kwargs*)

New in version 2.0. Abjad model of the oboe:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Oboe()(staff)
Oboe()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Oboe }
    \set Staff.shortInstrumentName = \markup { Ob. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The oboe targets staff context by default.

Read-only properties

Oboe.**default_instrument_name**
Read-only default instrument name.

Return string.

Oboe.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Oboe.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Oboe.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Oboe.**is_primary_instrument**

Oboe.**is_secondary_instrument**

Oboe.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Oboe.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Oboe.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Oboe.**storage_format**
Storage format of Abjad object.

Return string.

Oboe.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Oboe.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Oboe.**all_clefs**

Oboe.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Oboe.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Oboe.**pitch_range**

Oboe.**primary_clefs**

Oboe.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Oboe.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Oboe.**sounding_pitch_of_fingered_middle_c**

Methods

Oboe.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Oboe.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Oboe.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Oboe.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Oboe.**__call__**(**args*)

Oboe.**__copy__**(**args*)

Oboe.**__deepcopy__**(**args*)

Oboe.**__delattr__**(**args*)

Oboe.**__eq__**(*arg*)

Oboe.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Oboe.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Oboe.**__hash__**()

Oboe.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

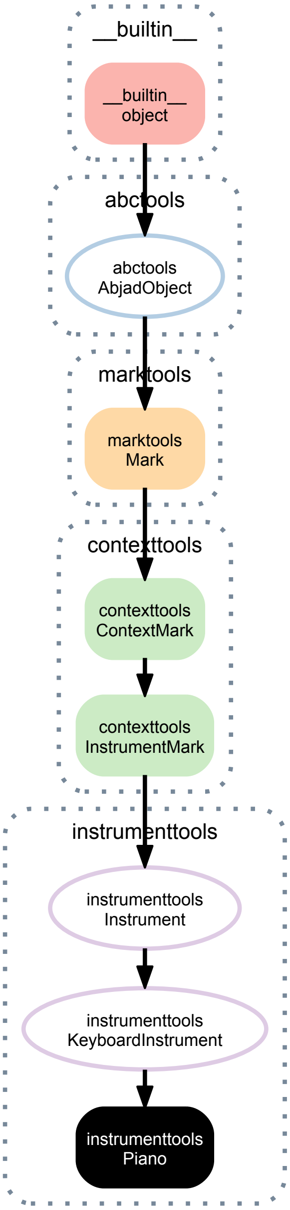
Oboe.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Oboe.**__ne__**(*arg*)

Oboe.**__repr__**()

8.2.34 instrumenttools.Piano



class `instrumenttools.`**`Piano`**(*target_context=None*, ***kwargs*)

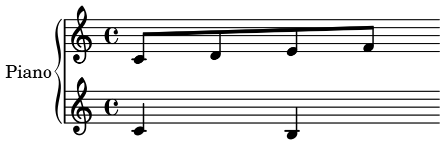
New in version 2.0. Abjad model of the piano:

```
>>> piano_staff = scoretools.PianoStaff([Staff("c'8 d'8 e'8 f'8"), Staff("c'4 b4")])
```

```
>>> instrumenttools.Piano()(piano_staff)
Piano()(PianoStaff<<2>>)
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \set PianoStaff.instrumentName = \markup { Piano }
    \set PianoStaff.shortInstrumentName = \markup { Pf. }
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        b4
    }
>>
```

```
>>> show(piano_staff)
```



The piano targets piano staff context by default.

Read-only properties

`Piano.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`Piano.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`Piano.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Piano.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`Piano.`**`is_primary_instrument`**

`Piano.`**`is_secondary_instrument`**

`Piano.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

Piano.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Piano.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Piano.**storage_format**

Storage format of Abjad object.

Return string.

Piano.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Piano.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Piano.**all_clefs**

Piano.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Piano.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Piano.**pitch_range**

Piano.**primary_clefs**

Piano.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Piano.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Piano.**sounding_pitch_of_fingered_middle_c**

Methods

Piano.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Piano.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Piano.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Piano.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Piano.**__call__**(**args*)

Piano.**__copy__**(**args*)

Piano.**__deepcopy__**(**args*)

Piano.**__delattr__**(**args*)

Piano.**__eq__**(*arg*)

Piano.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Piano.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Piano.**__hash__**()

Piano.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

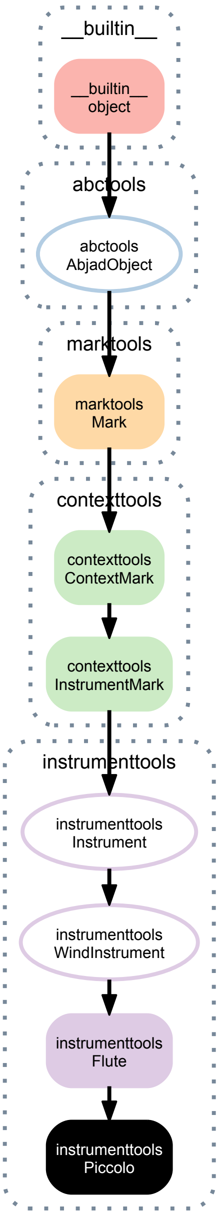
Piano.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Piano.**__ne__**(*arg*)

Piano.**__repr__**()

8.2.35 instrumenttools.Piccolo



class `instrumenttools.Piccolo` (***kwargs*)

New in version 2.0. Abjad model of the piccolo:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Piccolo()(staff)
Piccolo()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The piccolo targets staff context by default.

Read-only properties

Piccolo.**default_instrument_name**
Read-only default instrument name.

Return string.

Piccolo.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Piccolo.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Piccolo.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Piccolo.**is_primary_instrument**

Piccolo.**is_secondary_instrument**

Piccolo.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Piccolo.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Piccolo.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Piccolo.**storage_format**
Storage format of Abjad object.

Return string.

`Piccolo.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Piccolo.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Piccolo.`**`all_clefs`**

`Piccolo.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Piccolo.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Piccolo.`**`pitch_range`**

`Piccolo.`**`primary_clefs`**

`Piccolo.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Piccolo.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Piccolo.**sounding_pitch_of_fingered_middle_c**

Methods

Piccolo.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

Piccolo.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Piccolo.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

Piccolo.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

Piccolo.**__call__**(**args*)

Piccolo.**__copy__**(**args*)

Piccolo.**__deepcopy__**(**args*)

Piccolo.**__delattr__**(**args*)

Piccolo.**__eq__**(*arg*)

Piccolo.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Piccolo.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Piccolo.**__hash__**()

Piccolo.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

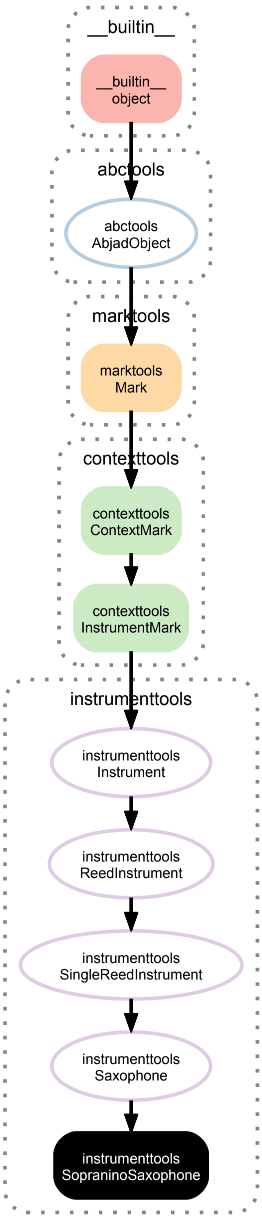
Piccolo.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Piccolo.**__ne__**(*arg*)

Piccolo.**__repr__**()

8.2.36 instrumenttools.SopraninoSaxophone



class instrumenttools.**SopraninoSaxophone**(***kwargs*)

New in version 2.6. Abjad model of the sopranino saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.SopraninoSaxophone()(staff)
SopraninoSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Sopranino saxophone }
    \set Staff.shortInstrumentName = \markup { Sopranino sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The sopranino saxophone is pitched in E-flat.

The sopranino saxophone targets staff context by default.

Read-only properties

SopraninoSaxophone.**default_instrument_name**

Read-only default instrument name.

Return string.

SopraninoSaxophone.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

SopraninoSaxophone.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

SopraninoSaxophone.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

SopraninoSaxophone.**is_primary_instrument**

SopraninoSaxophone.**is_secondary_instrument**

SopraninoSaxophone.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

SopraninoSaxophone.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`SopraninoSaxophone.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`SopraninoSaxophone.`**`storage_format`**

Storage format of Abjad object.

Return string.

`SopraninoSaxophone.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`SopraninoSaxophone.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`SopraninoSaxophone.`**`all_clefs`**

`SopraninoSaxophone.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`SopraninoSaxophone.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

SopraninoSaxophone.**pitch_range**

SopraninoSaxophone.**primary_clefs**

SopraninoSaxophone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

SopraninoSaxophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

SopraninoSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

SopraninoSaxophone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

SopraninoSaxophone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

SopraninoSaxophone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

SopraninoSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

SopraninoSaxophone.**__call__**(**args*)

SopraninoSaxophone.**__copy__**(**args*)

SopraninoSaxophone.**__deepcopy__**(**args*)

SopraninoSaxophone.**__delattr__**(**args*)

SopraninoSaxophone.**__eq__**(*arg*)

SopraninoSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopraninoSaxophone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SopraninoSaxophone.**__hash__**()

SopraninoSaxophone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

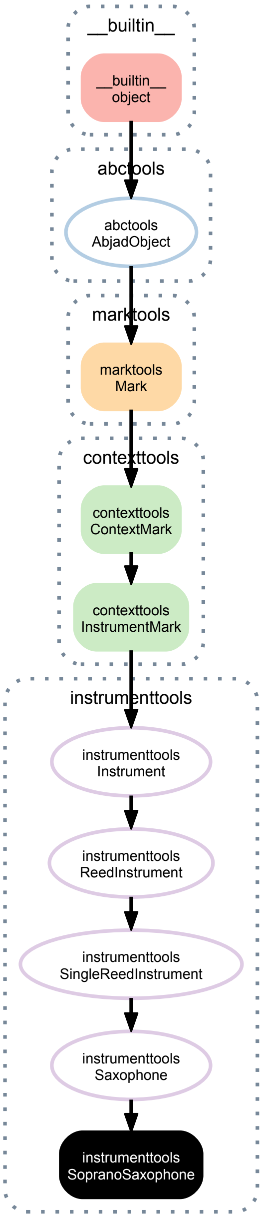
SopraninoSaxophone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopraninoSaxophone.**__ne__**(*arg*)

SopraninoSaxophone.**__repr__**()

8.2.37 instrumenttools.SopranoSaxophone



class instrumenttools.**SopranoSaxophone**(***kwargs*)

New in version 2.6. Abjad model of the soprano saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.SopranoSaxophone()(staff)
SopranoSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Soprano saxophone }
    \set Staff.shortInstrumentName = \markup { Sop. sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The soprano saxophone is pitched in B-flat.

The soprano saxophone targets staff context by default.

Read-only properties

SopranoSaxophone.**default_instrument_name**
Read-only default instrument name.

Return string.

SopranoSaxophone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

SopranoSaxophone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

SopranoSaxophone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

SopranoSaxophone.**is_primary_instrument**

SopranoSaxophone.**is_secondary_instrument**

SopranoSaxophone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

SopranoSaxophone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

SopranoSaxophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

SopranoSaxophone.**storage_format**
Storage format of Abjad object.

Return string.

`SopranoSaxophone.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`SopranoSaxophone.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`SopranoSaxophone.`**`all_clefs`**

`SopranoSaxophone.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`SopranoSaxophone.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`SopranoSaxophone.`**`pitch_range`**

`SopranoSaxophone.`**`primary_clefs`**

`SopranoSaxophone.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

SopranoSaxophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

SopranoSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

SopranoSaxophone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

SopranoSaxophone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

SopranoSaxophone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

SopranoSaxophone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

SopranoSaxophone.**__call__**(**args*)

SopranoSaxophone.**__copy__**(**args*)

SopranoSaxophone.**__deepcopy__**(**args*)

SopranoSaxophone.**__delattr__**(**args*)

SopranoSaxophone.**__eq__**(*arg*)

SopranoSaxophone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopranoSaxophone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SopranoSaxophone.**__hash__**()

SopranoSaxophone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

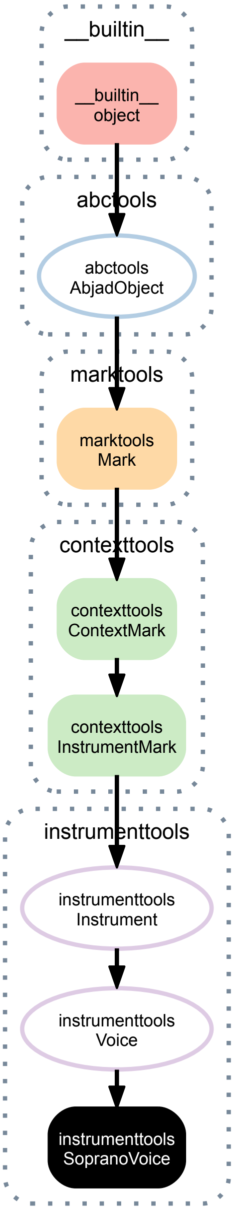
SopranoSaxophone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopranoSaxophone.**__ne__**(*arg*)

SopranoSaxophone.**__repr__**()

8.2.38 instrumenttools.SopranoVoice



class instrumenttools.**SopranoVoice**(***kwargs*)

New in version 2.8. Abjad model of the soprano voice:

```
>>> staff = Staff("c''8 d''8 e''8 f''8")
```

```
>>> instrumenttools.SopranoVoice()(staff)
SopranoVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Soprano voice }
    \set Staff.shortInstrumentName = \markup { Soprano }
    c''8
    d''8
    e''8
    f''8
}
```

```
>>> show(staff)
```

The soprano voice targets staff context by default.

Read-only properties

SopranoVoice.**default_instrument_name**

Read-only default instrument name.

Return string.

SopranoVoice.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

SopranoVoice.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

SopranoVoice.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

SopranoVoice.**is_primary_instrument**

SopranoVoice.**is_secondary_instrument**

SopranoVoice.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

SopranoVoice.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

SopranoVoice.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

SopranoVoice.**storage_format**

Storage format of Abjad object.

Return string.

SopranoVoice.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

SopranoVoice.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

SopranoVoice.**all_clefs**

SopranoVoice.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

SopranoVoice.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

SopranoVoice.**pitch_range**

SopranoVoice.**primary_clefs**

SopranoVoice.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

SopranoVoice.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

SopranoVoice.**sounding_pitch_of_fingered_middle_c**

Methods

SopranoVoice.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

SopranoVoice.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

SopranoVoice.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

SopranoVoice.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

SopranoVoice.**__call__**(**args*)

SopranoVoice.**__copy__**(**args*)

SopranoVoice.**__deepcopy__**(**args*)

SopranoVoice.**__delattr__**(**args*)

SopranoVoice.**__eq__**(*arg*)

SopranoVoice.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopranoVoice.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SopranoVoice.**__hash__**()

SopranoVoice.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

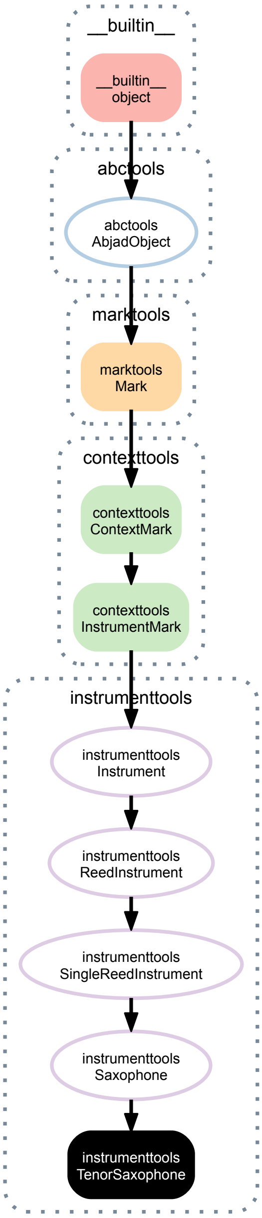
SopranoVoice.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SopranoVoice.**__ne__**(*arg*)

SopranoVoice.**__repr__**()

8.2.39 instrumenttools.TenorSaxophone



class `instrumenttools.`**`TenorSaxophone`**(***kwargs*)

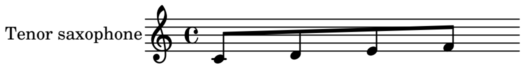
New in version 2.6. Abjad model of the tenor saxophone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.TenorSaxophone()(staff)
TenorSaxophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Tenor saxophone }
    \set Staff.shortInstrumentName = \markup { Ten. sax. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



The tenor saxophone is pitched in B-flat.

The tenor saxophone targets staff context by default.

Read-only properties

TenorSaxophone.**default_instrument_name**
Read-only default instrument name.

Return string.

TenorSaxophone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

TenorSaxophone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

TenorSaxophone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

TenorSaxophone.**is_primary_instrument**

TenorSaxophone.**is_secondary_instrument**

TenorSaxophone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

TenorSaxophone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

TenorSaxophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

TenorSaxophone.**storage_format**
Storage format of Abjad object.

Return string.

TenorSaxophone.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

TenorSaxophone.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

TenorSaxophone.**all_clefs**

TenorSaxophone.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

TenorSaxophone.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

TenorSaxophone.**pitch_range**

TenorSaxophone.**primary_clefs**

TenorSaxophone.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

TenorSaxophone.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

TenorSaxophone.**sounding_pitch_of_fingered_middle_c**

Methods

TenorSaxophone.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

TenorSaxophone.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

TenorSaxophone.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

TenorSaxophone.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

TenorSaxophone.**__call__**(**args*)

TenorSaxophone.**__copy__**(**args*)

TenorSaxophone.**__deepcopy__**(**args*)

TenorSaxophone.**__delattr__**(**args*)

TenorSaxophone.**__eq__**(*arg*)

TenorSaxophone.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`TenorSaxophone.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TenorSaxophone.__hash__()`

`TenorSaxophone.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

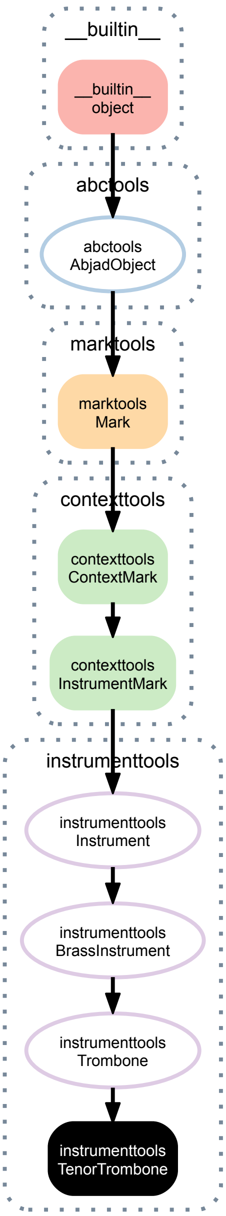
`TenorSaxophone.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TenorSaxophone.__ne__(arg)`

`TenorSaxophone.__repr__()`

8.2.40 instrumenttools.TenorTrombone



class instrumenttools.**TenorTrombone**(***kwargs*)

New in version 2.0. Abjad model of the tenor trombone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.TenorTrombone()(staff)
TenorTrombone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Tenor trombone }
    \set Staff.shortInstrumentName = \markup { Ten. trb. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The tenor trombone targets staff context by default.

Read-only properties

TenorTrombone.**default_instrument_name**

Read-only default instrument name.

Return string.

TenorTrombone.**default_short_instrument_name**

Read-only default short instrument name.

Return string.

TenorTrombone.**effective_context**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

TenorTrombone.**interval_of_transposition**

Read-only interval of transposition.

Return melodic diatonic interval.

TenorTrombone.**is_primary_instrument**

TenorTrombone.**is_secondary_instrument**

TenorTrombone.**is_transposing**

True when instrument is transposing. False otherwise.

Return boolean.

TenorTrombone.**lilypond_format**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`TenorTrombone.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`TenorTrombone.`**`storage_format`**

Storage format of Abjad object.

Return string.

`TenorTrombone.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`TenorTrombone.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`TenorTrombone.`**`all_clefs`**

`TenorTrombone.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`TenorTrombone.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

TenorTrombone.**pitch_range**

TenorTrombone.**primary_clefs**

TenorTrombone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

TenorTrombone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

TenorTrombone.**sounding_pitch_of_fingered_middle_c**

Methods

TenorTrombone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

TenorTrombone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

TenorTrombone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

TenorTrombone.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

TenorTrombone.**__call__**(**args*)

TenorTrombone.**__copy__**(**args*)

TenorTrombone.**__deepcopy__**(**args*)

TenorTrombone.**__delattr__**(**args*)

TenorTrombone.**__eq__**(*arg*)

TenorTrombone.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

TenorTrombone.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

TenorTrombone.**__hash__**()

TenorTrombone.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

TenorTrombone.**__lt__**(*expr*)

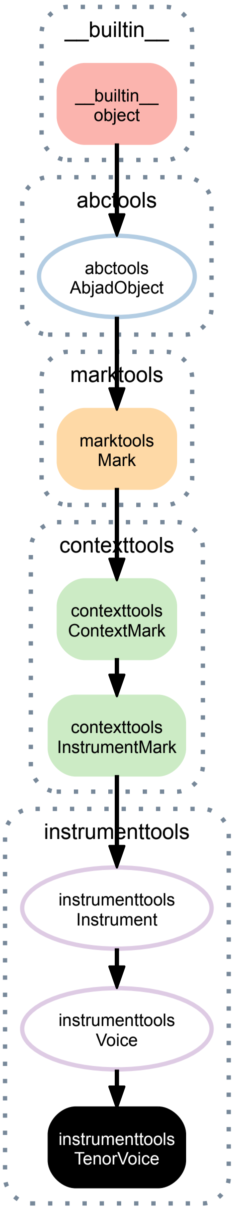
Abjad objects by default do not implement this method.

Raise exception.

TenorTrombone.**__ne__**(*arg*)

TenorTrombone.**__repr__**()

8.2.41 instrumenttools.TenorVoice



class instrumenttools.**TenorVoice**(***kwargs*)

New in version 2.8. Abjad model of the tenor voice:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.TenorVoice()(staff)
TenorVoice()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Tenor voice }
    \set Staff.shortInstrumentName = \markup { Tenor }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

The tenor voice targets staff context by default.

Read-only properties

`TenorVoice.default_instrument_name`
Read-only default instrument name.

Return string.

`TenorVoice.default_short_instrument_name`
Read-only default short instrument name.

Return string.

`TenorVoice.effective_context`
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`TenorVoice.interval_of_transposition`
Read-only interval of transposition.

Return melodic diatonic interval.

`TenorVoice.is_primary_instrument`

`TenorVoice.is_secondary_instrument`

`TenorVoice.is_transposing`
True when instrument is transposing. False otherwise.

Return boolean.

`TenorVoice.lilypond_format`
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`TenorVoice.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`TenorVoice.storage_format`
Storage format of Abjad object.

Return string.

`TenorVoice.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`TenorVoice.`**`traditional_pitch_range`**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`TenorVoice.`**`all_clefs`**

`TenorVoice.`**`instrument_name`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`TenorVoice.`**`instrument_name_markup`**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`TenorVoice.`**`pitch_range`**

`TenorVoice.`**`primary_clefs`**

`TenorVoice.`**`short_instrument_name`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

TenorVoice.**short_instrument_name_markup**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

TenorVoice.**sounding_pitch_of_fingered_middle_c**

Methods

TenorVoice.**attach**(*start_component*)

Make sure no context mark of same type is already attached to score component that starts with start component.

TenorVoice.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

TenorVoice.**get_default_performer_name**(*locale=None*)

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

TenorVoice.**get_performer_names**()

New in version 2.5. Get performer names.

Special methods

TenorVoice.**__call__**(**args*)

TenorVoice.**__copy__**(**args*)

TenorVoice.**__deepcopy__**(**args*)

TenorVoice.**__delattr__**(**args*)

TenorVoice.**__eq__**(*arg*)

TenorVoice.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`TenorVoice.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TenorVoice.__hash__()`

`TenorVoice.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

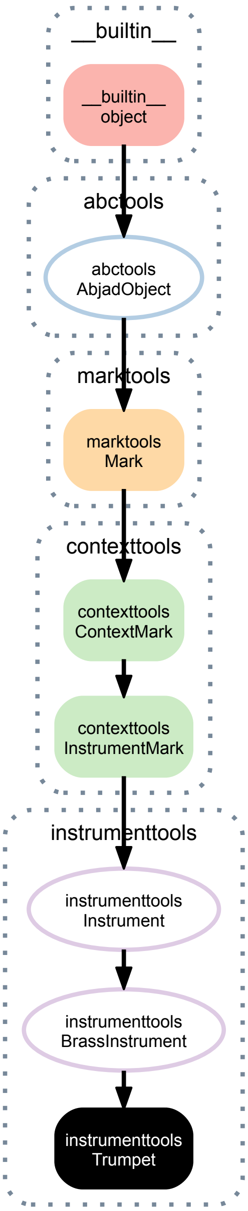
`TenorVoice.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TenorVoice.__ne__(arg)`

`TenorVoice.__repr__()`

8.2.42 instrumenttools.Trumpet



class `instrumenttools.`**`Trumpet`**`(**kwargs)`
New in version 2.0. Abjad model of the trumpet:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Trumpet()(staff)
Trumpet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Trumpet }
    \set Staff.shortInstrumentName = \markup { Tp. }
    c'8
    d'8
    e'8
    f'8
}
```

The trumpet targets staff context by default.

Read-only properties

`Trumpet.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`Trumpet.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`Trumpet.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Trumpet.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`Trumpet.`**`is_primary_instrument`**

`Trumpet.`**`is_secondary_instrument`**

`Trumpet.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`Trumpet.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Trumpet.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Trumpet.`**storage_format**

Storage format of Abjad object.

Return string.

`Trumpet.`**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`Trumpet.`**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`Trumpet.`**all_clefs**

`Trumpet.`**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`Trumpet.`**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`Trumpet.`**pitch_range**

`Trumpet.`**primary_clefs**

`Trumpet.`**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

`Trumpet.`**`short_instrument_name_markup`**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Trumpet.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Trumpet.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`Trumpet.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Trumpet.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Trumpet.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`Trumpet.`**`__call__`**`(`**args*`)`

`Trumpet.`**`__copy__`**`(`**args*`)`

`Trumpet.`**`__deepcopy__`**`(`**args*`)`

Trumpet.**__delattr__**(**args*)

Trumpet.**__eq__**(*arg*)

Trumpet.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Trumpet.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Trumpet.**__hash__**()

Trumpet.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

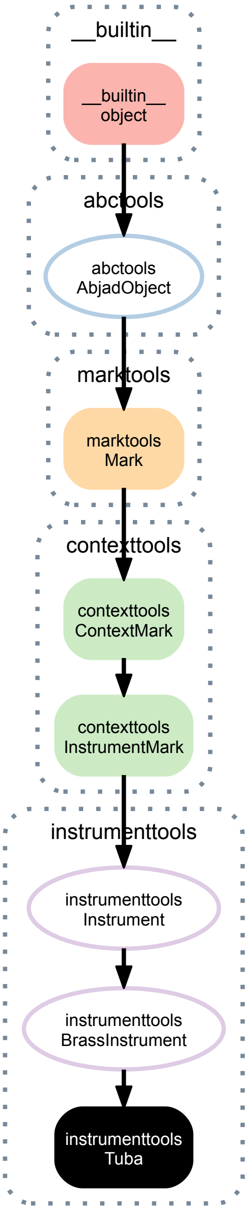
Trumpet.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Trumpet.**__ne__**(*arg*)

Trumpet.**__repr__**()

8.2.43 instrumenttools.Tuba



class `instrumenttools.Tuba`(***kwargs*)

New in version 2.0. Abjad model of the tuba:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('bass')(staff)
ClefMark('bass')(Staff{4})
```

```
>>> instrumenttools.Tuba()(staff)
Tuba()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    \set Staff.instrumentName = \markup { Tuba }
    \set Staff.shortInstrumentName = \markup { Tb. }
    c'8
    d'8
    e'8
    f'8
}
```

The tuba targets staff context by default.

Read-only properties

Tuba.**default_instrument_name**
Read-only default instrument name.

Return string.

Tuba.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Tuba.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Tuba.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Tuba.**is_primary_instrument**

Tuba.**is_secondary_instrument**

Tuba.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Tuba.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Tuba.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Tuba.**storage_format**
Storage format of Abjad object.

Return string.

Tuba.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Tuba.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Tuba.**all_clefs**

Tuba.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Tuba.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Tuba.**pitch_range**

Tuba.**primary_clefs**

Tuba.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Tuba.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Tuba.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Tuba.`**`attach`**`(`*start_component*`)`

Make sure no context mark of same type is already attached to score component that starts with start component.

`Tuba.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Tuba.`**`get_default_performer_name`**`(`*locale=None*`)`

New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Tuba.`**`get_performer_names`**`()`

New in version 2.5. Get performer names.

Special methods

`Tuba.`**`__call__`**`(`**args*`)`

`Tuba.`**`__copy__`**`(`**args*`)`

`Tuba.`**`__deepcopy__`**`(`**args*`)`

`Tuba.`**`__delattr__`**`(`**args*`)`

`Tuba.`**`__eq__`**`(`*arg*`)`

`Tuba.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`Tuba.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

Tuba.**__hash__**()

Tuba.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

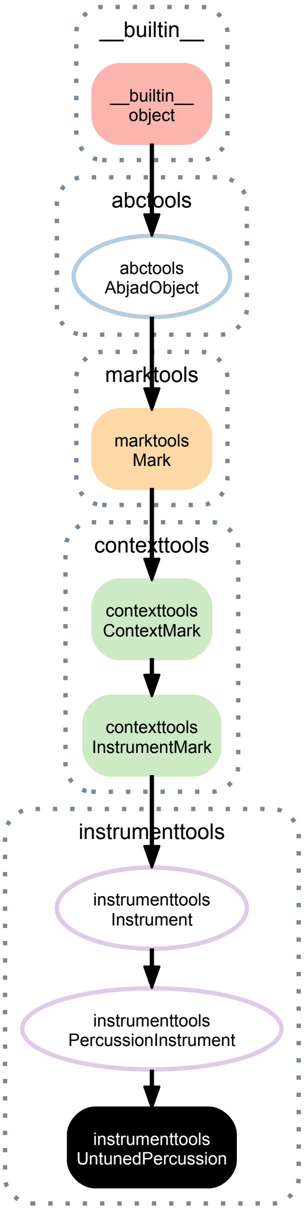
Tuba.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Tuba.**__ne__**(*arg*)

Tuba.**__repr__**()

8.2.44 instrumenttools.UntunedPercussion



class instrumenttools.**UntunedPercussion**(***kwargs*)
New in version 2.0. Abjad model of untuned percussion:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.UntunedPercussion()(staff)
UntunedPercussion()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Untuned percussion }
    \set Staff.shortInstrumentName = \markup { Perc. }
    c'8
    d'8
    e'8
    f'8
}
```

Untuned percussion targets the staff context by default.

Read-only properties

`UntunedPercussion.`**`default_instrument_name`**
Read-only default instrument name.

Return string.

`UntunedPercussion.`**`default_short_instrument_name`**
Read-only default short instrument name.

Return string.

`UntunedPercussion.`**`effective_context`**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`UntunedPercussion.`**`interval_of_transposition`**
Read-only interval of transposition.

Return melodic diatonic interval.

`UntunedPercussion.`**`is_primary_instrument`**

`UntunedPercussion.`**`is_secondary_instrument`**

`UntunedPercussion.`**`is_transposing`**
True when instrument is transposing. False otherwise.

Return boolean.

`UntunedPercussion.`**`lilypond_format`**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`UntunedPercussion.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`UntunedPercussion.`**storage_format**

Storage format of Abjad object.

Return string.

`UntunedPercussion.`**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

`UntunedPercussion.`**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

`UntunedPercussion.`**all_clefs**

`UntunedPercussion.`**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

`UntunedPercussion.`**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

`UntunedPercussion.`**pitch_range**

`UntunedPercussion.`**primary_clefs**

`UntunedPercussion.`**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

UntunedPercussion.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

UntunedPercussion.**sounding_pitch_of_fingered_middle_c**

Methods

UntunedPercussion.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

UntunedPercussion.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

UntunedPercussion.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

UntunedPercussion.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

UntunedPercussion.**__call__**(**args*)

UntunedPercussion.**__copy__**(**args*)

UntunedPercussion.**__deepcopy__**(**args*)

UntunedPercussion.**__delattr__**(**args*)

UntunedPercussion.**__eq__**(*arg*)

UntunedPercussion.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

UntunedPercussion.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

UntunedPercussion.**__hash__**()

UntunedPercussion.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

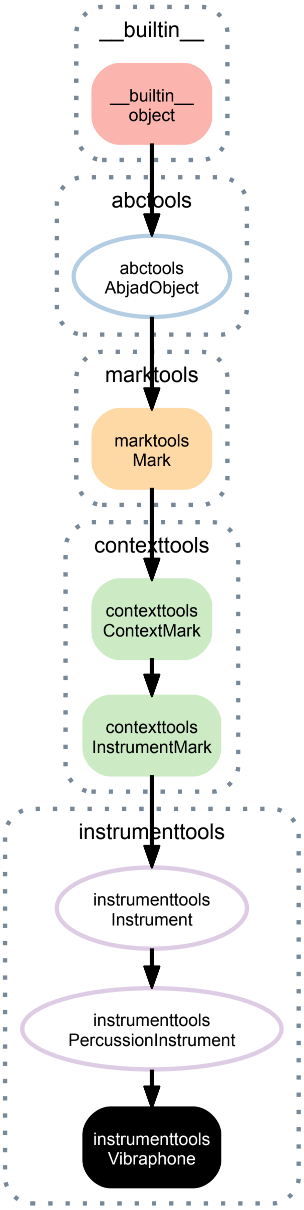
UntunedPercussion.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

UntunedPercussion.**__ne__**(*arg*)

UntunedPercussion.**__repr__**()

8.2.45 instrumenttools.Vibraphone



class `instrumenttools.Vibraphone(**kwargs)`

New in version 2.0. Abjad model of the vibraphone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Vibraphone()(staff)
Vibraphone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Vibraphone }
    \set Staff.shortInstrumentName = \markup { Vibr. }
    c'8
    d'8
    e'8
    f'8
}
```

The vibraphone targets staff context by default.

Read-only properties

`Vibraphone.default_instrument_name`
Read-only default instrument name.

Return string.

`Vibraphone.default_short_instrument_name`
Read-only default short instrument name.

Return string.

`Vibraphone.effective_context`
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Vibraphone.interval_of_transposition`
Read-only interval of transposition.

Return melodic diatonic interval.

`Vibraphone.is_primary_instrument`

`Vibraphone.is_secondary_instrument`

`Vibraphone.is_transposing`
True when instrument is transposing. False otherwise.

Return boolean.

`Vibraphone.lilypond_format`
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Vibraphone.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Vibraphone.storage_format`
Storage format of Abjad object.

Return string.

`Vibraphone.target_context`
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Vibraphone.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Vibraphone.**all_clefs**

Vibraphone.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Vibraphone.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Vibraphone.**pitch_range**

Vibraphone.**primary_clefs**

Vibraphone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Vibraphone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Vibraphone.**sounding_pitch_of_fingered_middle_c**

Methods

Vibraphone.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Vibraphone.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Vibraphone.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Vibraphone.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Vibraphone.**__call__**(**args*)

Vibraphone.**__copy__**(**args*)

Vibraphone.**__deepcopy__**(**args*)

Vibraphone.**__delattr__**(**args*)

Vibraphone.**__eq__**(*arg*)

Vibraphone.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Vibraphone.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Vibraphone.**__hash__**()

Vibraphone.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

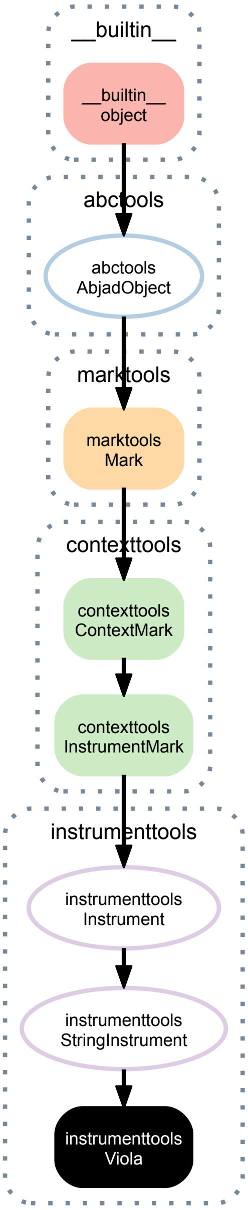
Vibraphone.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Vibraphone.**__ne__**(*arg*)

Vibraphone.**__repr__**()

8.2.46 instrumenttools.Viola



class instrumenttools.**Viola**(***kwargs*)
New in version 2.0. Abjad model of the viola:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('alto')(staff)
ClefMark('alto')(Staff{4})
```

```
>>> instrumenttools.Viola()(staff)
Viola()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "alto"
    \set Staff.instrumentName = \markup { Viola }
    \set Staff.shortInstrumentName = \markup { Va. }
    c'8
    d'8
    e'8
    f'8
}
```

The viola targets staff context by default.

Read-only properties

Viola.**default_instrument_name**
Read-only default instrument name.

Return string.

Viola.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Viola.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Viola.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Viola.**is_primary_instrument**

Viola.**is_secondary_instrument**

Viola.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Viola.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Viola.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Viola.**storage_format**

Storage format of Abjad object.

Return string.

Viola.**target_context**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Viola.**traditional_pitch_range**

Read-only traditional pitch range.

Return pitch range.

Read/write properties

Viola.**all_clefs**

Viola.**instrument_name**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Viola.**instrument_name_markup**

Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Viola.**pitch_range**

Viola.**primary_clefs**

Viola.**short_instrument_name**

Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Viola.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Viola.**sounding_pitch_of_fingered_middle_c**

Methods

Viola.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Viola.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Viola.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Viola.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Viola.**__call__**(*args*)

Viola.**__copy__**(*args*)

Viola.**__deepcopy__**(*args*)

Viola.**__delattr__**(**args*)

Viola.**__eq__**(*arg*)

Viola.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Viola.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Viola.**__hash__**()

Viola.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

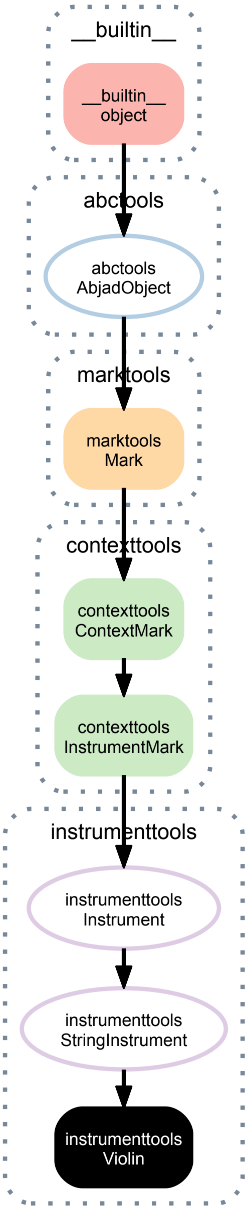
Viola.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Viola.**__ne__**(*arg*)

Viola.**__repr__**()

8.2.47 instrumenttools.Violin



class `instrumenttools.Violin`(***kwargs*)

New in version 2.0. Abjad model of the violin:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Violin }
    \set Staff.shortInstrumentName = \markup { Vn. }
    c'8
    d'8
    e'8
    f'8
}
```

The violin targets staff context by default.

Read-only properties

`Violin.`**`default_instrument_name`**

Read-only default instrument name.

Return string.

`Violin.`**`default_short_instrument_name`**

Read-only default short instrument name.

Return string.

`Violin.`**`effective_context`**

Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

`Violin.`**`interval_of_transposition`**

Read-only interval of transposition.

Return melodic diatonic interval.

`Violin.`**`is_primary_instrument`**

`Violin.`**`is_secondary_instrument`**

`Violin.`**`is_transposing`**

True when instrument is transposing. False otherwise.

Return boolean.

`Violin.`**`lilypond_format`**

Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

`Violin.`**`start_component`**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Violin.`**`storage_format`**

Storage format of Abjad object.

Return string.

`Violin.`**`target_context`**

Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Violin.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Violin.**all_clefs**

Violin.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Violin.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Violin.**pitch_range**

Violin.**primary_clefs**

Violin.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Violin.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

Violin.**sounding_pitch_of_fingered_middle_c**

Methods

Violin.**attach**(*start_component*)
Make sure no context mark of same type is already attached to score component that starts with start component.

Violin.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

Violin.**get_default_performer_name**(*locale=None*)
New in version 2.5. Get default player name.

Available values for *locale* are 'en-us' and 'en-uk'.

Violin.**get_performer_names**()
New in version 2.5. Get performer names.

Special methods

Violin.**__call__**(**args*)

Violin.**__copy__**(**args*)

Violin.**__deepcopy__**(**args*)

Violin.**__delattr__**(**args*)

Violin.**__eq__**(*arg*)

Violin.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Violin.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Violin.**__hash__**()

Violin.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

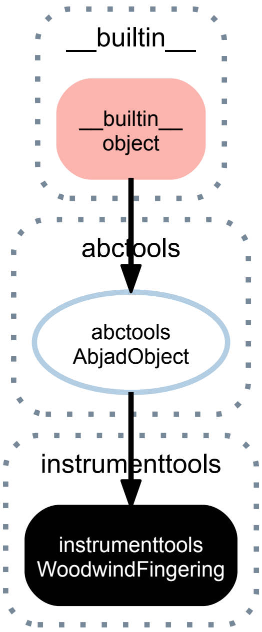
Violin.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Violin.**__ne__**(*arg*)

Violin.**__repr__**()

8.2.48 instrumenttools.WoodwindFingering



class instrumenttools.**WoodwindFingering**(*arg*, *center_column=None*, *left_hand=None*, *right_hand=None*)
Abjad model of a woodwind fingering, inspired by Mike Solomon's LilyPond woodwind diagrams.

Initialize from a valid instrument name and up to three keyword lists or tuples:

```
>>> center_column = ('one', 'two', 'three', 'five')
>>> left_hand = ('R', 'thumb')
>>> right_hand = ('e',)
>>> ww = instrumenttools.WoodwindFingering('clarinet', center_column = center_column, left_hand = left_h
>>> ww
WoodwindFingering('clarinet', center_column=('one', 'two', 'three', 'five'), left_hand=('R', 'thumb'), r
```

Initialize a WoodwindFingering from another WoodwindFingering:

```
>>> ww2 = instrumenttools.WoodwindFingering(ww)
>>> ww2
WoodwindFingering('clarinet', center_column=('one', 'two', 'three', 'five'), left_hand=('R', 'thumb'), r
```

Call a WoodwindFingering to create a woodwind diagram MarkupCommand:

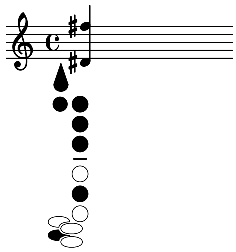
```
>>> fingering_command = ww()
>>> fingering_command
MarkupCommand('woodwind-diagram', Scheme('clarinet'), Scheme([SchemePair(('cc', ('one', 'two', 'three',
```

Attach the MarkupCommand to score components, such as a chord representing a multiphonic sound:

```
>>> markup = markuptools.Markup(fingering_command, direction=Down)
>>> chord = Chord("<ds' fs''>4")
>>> markup.attach(chord)
Markup((MarkupCommand('woodwind-diagram', Scheme('clarinet'), Scheme([SchemePair(('cc', ('one', 'two', '
```

```
>>> f(chord)
<ds' fs''>4
        _ \markup {
                \woodwind-diagram
                        #'clarinet
                        #'((cc . (one two three five)) (lh . (R thumb)) (rh . (e)))
                }
```

```
>>> show(chord)
```



Initialize fingerings for eight different woodwind instruments:

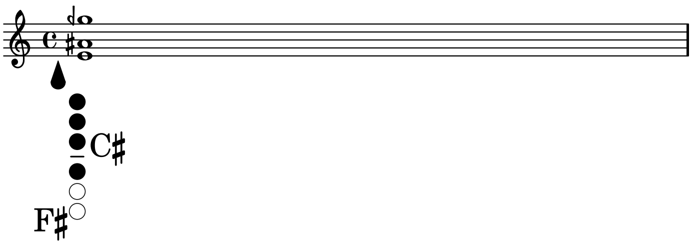
```
>>> instrument_names = ['piccolo', 'flute', 'oboe', 'clarinet', 'bass-clarinet', 'saxophone', 'bassoon',
>>> for name in instrument_names:
...     instrumenttools.WoodwindFingering(name)
...
WoodwindFingering('piccolo', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('flute', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('oboe', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('bass-clarinet', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('saxophone', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('bassoon', center_column=(), left_hand=(), right_hand=())
WoodwindFingering('contrabassoon', center_column=(), left_hand=(), right_hand=())
```

An override displays diagrams symbolically instead of graphically:

```
>>> chord = Chord("e' as' gqf''", (1,1))
>>> fingering = instrumenttools.WoodwindFingering('clarinet', center_column=['one', 'two', 'three', 'fou
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand('override', schemetools.SchemePair('graphical', False))
>>> markup = markuptools.Markup([not_graphical, diagram], direction=Down)
>>> markup.attach(chord)
Markup((MarkupCommand('override', SchemePair(('graphical', False))), MarkupCommand('woodwind-diagram', S
```

```
>>> f(chord)
<e' as' gqf''>1
        _ \markup {
                \override
                        #'(graphical . #f)
                \woodwind-diagram
                        #'clarinet
                        #'((cc . (one two three four)) (lh . (R cis)) (rh . (fis)))
                }
```

```
>>> show(chord)
```

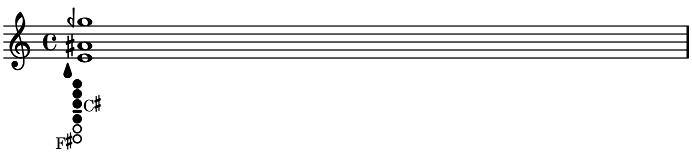


The thickness and size of diagrams can also be changed with overrides:

```
>>> chord = Chord("e' as' gqf''", (1,1))
>>> fingering = instrumenttools.WoodwindFingering('clarinet', center_column=['one', 'two', 'three', 'fou
>>> diagram = fingering()
>>> not_graphical = markuptools.MarkupCommand('override', schemetools.SchemePair('graphical', False))
>>> size = markuptools.MarkupCommand('override', schemetools.SchemePair('size', .5))
>>> thickness = markuptools.MarkupCommand('override', schemetools.SchemePair('thickness', .4))
>>> markup = markuptools.Markup([not_graphical, size, thickness, diagram], direction=Down)
>>> markup.attach(chord)
Markup((MarkupCommand('override', SchemePair(('graphical', False))), MarkupCommand('override', SchemePai
```

```
>>> f(chord)
<e' as' gqf''>1
            _ \markup {
                        \override
                                    #'(graphical . #f)
                        \override
                                    #'(size . 0.5)
                        \override
                                    #'(thickness . 0.4)
                        \woodwind-diagram
                                    #'clarinet
                                    #'((cc . (one two three four)) (lh . (R cis)) (rh . (fis)))
                        }
```

```
>>> show(chord)
```



Return woodwind fingering.

Read-only properties

`WoodwindFingering.`**`center_column`**

Read-only tuple of contents of key strings in center column key group:

```
>>> ww.center_column
('one', 'two', 'three', 'five')
```

Return tuple.

`WoodwindFingering.`**`instrument_name`**

Read-only string of valid woodwind instrument name:

```
>>> ww.instrument_name
'clarinet'
```

Return string.

`WoodwindFingering.`**`left_hand`**

Read-only tuple of contents of key strings in left hand key group:

```
>>> ww.left_hand
('R', 'thumb')
```

Return tuple.

WoodwindFingering.**right_hand**
Read-only tuple of contents of key strings in right hand key group:

```
>>> ww.right_hand
('e',)
```

Return tuple.

WoodwindFingering.**storage_format**
Storage format of Abjad object.

Return string.

Methods

WoodwindFingering.**print_guide**()
Print read-only string containing instrument's valid key strings, instrument diagram, and syntax explanation.

Return string.

Special methods

WoodwindFingering.**__call__**()

WoodwindFingering.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

WoodwindFingering.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

WoodwindFingering.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

WoodwindFingering.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

WoodwindFingering.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

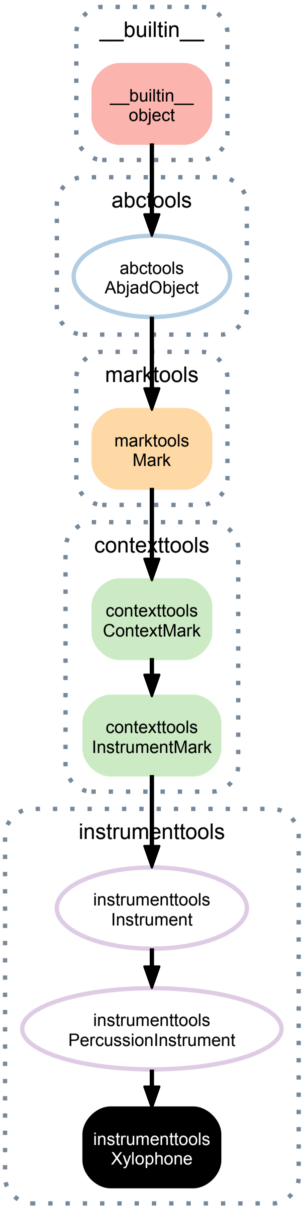
WoodwindFingering.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

WoodwindFingering.**__repr__**()
Interpreter representation of Abjad object.

Return string.

8.2.49 instrumenttools.Xylophone



class instrumenttools.**Xylophone**(***kwargs*)

New in version 2.0. Abjad model of the xylphone:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> instrumenttools.Xylophone()(staff)
Xylophone()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Xylophone }
    \set Staff.shortInstrumentName = \markup { Xyl. }
    c'8
    d'8
    e'8
    f'8
}
```

The xylophone targets staff context by default.

Read-only properties

Xylophone.**default_instrument_name**
Read-only default instrument name.

Return string.

Xylophone.**default_short_instrument_name**
Read-only default short instrument name.

Return string.

Xylophone.**effective_context**
Read-only reference to effective context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.effective_context is None
True
```

Return context mark or none.

Xylophone.**interval_of_transposition**
Read-only interval of transposition.

Return melodic diatonic interval.

Xylophone.**is_primary_instrument**

Xylophone.**is_secondary_instrument**

Xylophone.**is_transposing**
True when instrument is transposing. False otherwise.

Return boolean.

Xylophone.**lilypond_format**
Read-only LilyPond input format of instrument mark:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.lilypond_format
['\set Staff.instrumentName = \markup { Flute }',
    '\set Staff.shortInstrumentName = \markup { Fl. }']
```

Return list.

Xylophone.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Xylophone.**storage_format**
Storage format of Abjad object.

Return string.

Xylophone.**target_context**
Read-only reference to target context of context mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.target_context is None
True
```

Return context mark or none.

Xylophone.**traditional_pitch_range**
Read-only traditional pitch range.

Return pitch range.

Read/write properties

Xylophone.**all_clefs**

Xylophone.**instrument_name**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name
'Flute'
```

Set instrument name:

```
>>> instrument.instrument_name = 'Alto Flute'
>>> instrument.instrument_name
'Alto Flute'
```

Return string.

Xylophone.**instrument_name_markup**
Get instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.instrument_name_markup
Markup(('Flute',))
```

Set instrument name:

```
>>> instrument.instrument_name_markup = 'Alto Flute'
>>> instrument.instrument_name_markup
Markup(('Alto Flute',))
```

Return markup.

Xylophone.**pitch_range**

Xylophone.**primary_clefs**

Xylophone.**short_instrument_name**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name
'Fl.'
```

Set short instrument name:

```
>>> instrument.short_instrument_name = 'Alto Fl.'
>>> instrument.short_instrument_name
'Alto Fl.'
```

Return string.

Xylophone.**short_instrument_name_markup**
Get short instrument name:

```
>>> instrument = contexttools.InstrumentMark('Flute', 'Fl.')
>>> instrument.short_instrument_name_markup
Markup(('Fl.',))
```

Set short instrument name:

```
>>> instrument.short_instrument_name_markup = 'Alto Fl.'
>>> instrument.short_instrument_name_markup
Markup(('Alto Fl.',))
```

Return markup.

`Xylophone.`**`sounding_pitch_of_fingered_middle_c`**

Methods

`Xylophone.`**`attach`**`(`*start_component*`)`
Make sure no context mark of same type is already attached to score component that starts with start component.

`Xylophone.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> context_mark = contexttools.ContextMark()(note)
```

```
>>> context_mark.start_component
Note("c'4")
```

```
>>> context_mark.detach()
ContextMark()
```

```
>>> context_mark.start_component is None
True
```

Return context mark.

`Xylophone.`**`get_default_performer_name`**`(`*locale=None*`)`
New in version 2.5. Get default player name.

Available values for *locale* are `'en-us'` and `'en-uk'`.

`Xylophone.`**`get_performer_names`**`()`
New in version 2.5. Get performer names.

Special methods

`Xylophone.`**`__call__`**`(`**args*`)`

`Xylophone.`**`__copy__`**`(`**args*`)`

`Xylophone.`**`__deepcopy__`**`(`**args*`)`

`Xylophone.`**`__delattr__`**`(`**args*`)`

`Xylophone.`**`__eq__`**`(`*arg*`)`

`Xylophone.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Xylophone.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Xylophone.`**`__hash__`**`()`

`Xylophone.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Xylophone.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Xylophone.__ne__(arg)`

`Xylophone.__repr__()`

8.3 Functions

8.3.1 instrumenttools.default_instrument_name_to_instrument_class

`instrumenttools.default_instrument_name_to_instrument_class(default_instrument_name)`
New in version 2.5. Change *default_instrument_name* to class name:

```
>>> instrumenttools.default_instrument_name_to_instrument_class('clarinet in E-flat')
<class 'abjad.tools.instrumenttools.EFlatClarinet.EFlatClarinet.EFlatClarinet'>
```

Return class.

When *default_instrument_name* matches no instrument class:

```
>>> instrumenttools.default_instrument_name_to_instrument_class('foo') is None
True
```

Return none.

8.3.2 instrumenttools.iterate_notes_and_chords_in_expr_outside_traditional_instrument_ra

`instrumenttools.iterate_notes_and_chords_in_expr_outside_traditional_instrument_ranges(e`
New in version 2.0. Iterate notes and chords in *expr* outside traditional instrument ranges:

```
>>> staff = Staff("c'8 r8 <d fs>8 r8")
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> list(
... instrumenttools.iterate_notes_and_chords_in_expr_outside_traditional_instrument_ranges(
... staff))
[Chord('<d fs>8')]
```

Return generator.

8.3.3 instrumenttools.list_instrument_names

`instrumenttools.list_instrument_names()`
New in version 2.5. List instrument names:

```
>>> for instrument_name in instrumenttools.list_instrument_names():
...     instrument_name
...
'accordion'
'alto flute'
'alto saxophone'
'alto trombone'
'clarinet in B-flat'
'baritone saxophone'
```

```
'baritone voice'
'bass clarinet'
'bass flute'
'bass saxophone'
'bass trombone'
'bass voice'
'bassoon'
'cello'
'clarinet in A'
'contrabass'
'contrabass clarinet'
'contrabass flute'
'contrabass saxophone'
'contrabassoon'
'contralto voice'
'clarinet in E-flat'
'English horn'
'flute'
'horn'
'glockenspiel'
'guitar'
'harp'
'harpsichord'
'marimba'
'mezzo-soprano voice'
'oboe'
'piano'
'piccolo'
'sopranino saxophone'
'soprano saxophone'
'soprano voice'
'tenor saxophone'
'tenor trombone'
'tenor voice'
'trumpet'
'tuba'
'untuned percussion'
'vibraphone'
'viola'
'violin'
'xylophone'
```

Return list.

8.3.4 instrumenttools.list_instruments

instrumenttools.**list_instruments**(*klasses=None*)

New in version 2.5. List instruments in `instrumenttools` module:

```
>>> for instrument in instrumenttools.list_instruments()[:5]:
...     instrument
...
<class 'abjad.tools.instrumenttools.Accordion.Accordion.Accordion'>
<class 'abjad.tools.instrumenttools.AltoFlute.AltoFlute.AltoFlute'>
<class 'abjad.tools.instrumenttools.AltoSaxophone.AltoSaxophone.AltoSaxophone'>
<class 'abjad.tools.instrumenttools.AltoTrombone.AltoTrombone.AltoTrombone'>
<class 'abjad.tools.instrumenttools.BFlatClarinet.BFlatClarinet.BFlatClarinet'>
```

Return list.

8.3.5 instrumenttools.list_primary_instruments

instrumenttools.**list_primary_instruments**()

New in version 2.5. List primary instruments:

```
>>> for primary_instrument in instrumenttools.list_primary_instruments():
...     primary_instrument
```

```
...
<class 'abjad.tools.instrumenttools.Accordion.Accordion.Accordion'>
<class 'abjad.tools.instrumenttools.AltoSaxophone.AltoSaxophone.AltoSaxophone'>
<class 'abjad.tools.instrumenttools.BFlatClarinet.BFlatClarinet.BFlatClarinet'>
<class 'abjad.tools.instrumenttools.BaritoneVoice.BaritoneVoice.BaritoneVoice'>
<class 'abjad.tools.instrumenttools.BassVoice.BassVoice.BassVoice'>
<class 'abjad.tools.instrumenttools.Bassoon.Bassoon.Bassoon'>
<class 'abjad.tools.instrumenttools.Cello.Cello.Cello'>
<class 'abjad.tools.instrumenttools.Contrabass.Contrabass.Contrabass'>
<class 'abjad.tools.instrumenttools.ContraltoVoice.ContraltoVoice.ContraltoVoice'>
<class 'abjad.tools.instrumenttools.Flute.Flute.Flute'>
<class 'abjad.tools.instrumenttools.FrenchHorn.FrenchHorn.FrenchHorn'>
<class 'abjad.tools.instrumenttools.Guitar.Guitar.Guitar'>
<class 'abjad.tools.instrumenttools.Harp.Harp.Harp'>
<class 'abjad.tools.instrumenttools.Harpsichord.Harpsichord.Harpsichord'>
<class 'abjad.tools.instrumenttools.MezzoSopranoVoice.MezzoSopranoVoice.MezzoSopranoVoice'>
<class 'abjad.tools.instrumenttools.Oboe.Oboe.Oboe'>
<class 'abjad.tools.instrumenttools.Piano.Piano.Piano'>
<class 'abjad.tools.instrumenttools.SopranoVoice.SopranoVoice.SopranoVoice'>
<class 'abjad.tools.instrumenttools.TenorTrombone.TenorTrombone.TenorTrombone'>
<class 'abjad.tools.instrumenttools.TenorVoice.TenorVoice.TenorVoice'>
<class 'abjad.tools.instrumenttools.Trumpet.Trumpet.Trumpet'>
<class 'abjad.tools.instrumenttools.Tuba.Tuba.Tuba'>
<class 'abjad.tools.instrumenttools.Viola.Viola.Viola'>
<class 'abjad.tools.instrumenttools.Violin.Violin.Violin'>
```

Return list

8.3.6 instrumenttools.list_secondary_instruments

instrumenttools.**list_secondary_instruments**()

New in version 2.5. List secondary instruments:

```
>>> for secondary_instrument in instrumenttools.list_secondary_instruments()[:5]:
...     secondary_instrument
...
<class 'abjad.tools.instrumenttools.AltoFlute.AltoFlute.AltoFlute'>
<class 'abjad.tools.instrumenttools.AltoTrombone.AltoTrombone.AltoTrombone'>
<class 'abjad.tools.instrumenttools.BaritoneSaxophone.BaritoneSaxophone.BaritoneSaxophone'>
<class 'abjad.tools.instrumenttools.BassClarinet.BassClarinet.BassClarinet'>
<class 'abjad.tools.instrumenttools.BassFlute.BassFlute.BassFlute'>
```

Return list

8.3.7 instrumenttools.notes_and_chords_in_expr_are_on_expected_clefs

instrumenttools.**notes_and_chords_in_expr_are_on_expected_clefs**(*expr*, *percussion_clef_is_allowed=True*)

New in version 2.0. True when notes and chords in *expr* are on expected clefs:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('treble')(staff)
ClefMark('treble')(Staff{4})
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> instrumenttools.notes_and_chords_in_expr_are_on_expected_clefs(staff)
True
```

False otherwise:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('alto')(staff)
ClefMark('alto')(Staff{4})
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> instrumenttools.notes_and_chords_in_expr_are_on_expected_clefs(staff)
False
```

Allow percussion clef when *percussion_clef_is_allowed* is true:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.ClefMark('percussion')(staff)
ClefMark('percussion')(Staff{4})
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \clef "percussion"
    \set Staff.instrumentName = \markup { Violin }
    \set Staff.shortInstrumentName = \markup { Vn. }
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> instrumenttools.notes_and_chords_in_expr_are_on_expected_clefs(
...     staff, percussion_clef_is_allowed=True)
True
```

Disallow percussion clef when *percussion_clef_is_allowed* is false:

```
>>> instrumenttools.notes_and_chords_in_expr_are_on_expected_clefs(
...     staff, percussion_clef_is_allowed=False)
False
```

Return boolean.

8.3.8 instrumenttools.notes_and_chords_in_expr_are_within_traditional_instrument_ranges

instrumenttools.**notes_and_chords_in_expr_are_within_traditional_instrument_ranges**(*expr*)

New in version 2.0. True when notes and chords in *expr* are within traditional instrument ranges:

```
>>> staff = Staff("c'8 r8 <d' fs'>8 r8")
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> instrumenttools.notes_and_chords_in_expr_are_within_traditional_instrument_ranges(
...     staff)
True
```

False otherwise:

```
>>> staff = Staff("c'8 r8 <d fs>8 r8")
>>> instrumenttools.Violin()(staff)
Violin()(Staff{4})
```

```
>>> instrumenttools.notes_and_chords_in_expr_are_within_traditional_instrument_ranges(
...     staff)
False
```

Return boolean.

8.3.9 instrumenttools.transpose_from_fingered_pitch_to_sounding_pitch

instrumenttools.**transpose_from_fingered_pitch_to_sounding_pitch**(*expr*)

New in version 2.0. Transpose notes and chords in *expr* from sounding pitch to fingered pitch:

```
>>> staff = Staff("<c' e' g'>4 d'4 r4 e'4")
>>> instrumenttools.BFlatClarinet()(staff)
BFlatClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in B-flat }
    \set Staff.shortInstrumentName = \markup { Cl. in B-flat }
    <c' e' g'>4
    d'4
    r4
    e'4
}
```

```
>>> for leaf in staff.leaves:
...   leaf.written_pitch_indication_is_at_sounding_pitch = False

>>> instrumenttools.transpose_from_fingered_pitch_to_sounding_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in B-flat }
    \set Staff.shortInstrumentName = \markup { Cl. in B-flat }
    <bf d' f'>4
    c'4
    r4
    d'4
}
```

Return none.

8.3.10 instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch

instrumenttools.**transpose_from_sounding_pitch_to_fingered_pitch**(*expr*)

New in version 2.0. Transpose notes and chords in *expr* from sounding pitch to fingered pitch:

```
>>> staff = Staff("<c' e' g'>4 d'4 r4 e'4")
>>> instrumenttools.BFlatClarinet()(staff)
BFlatClarinet()(Staff{4})
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in B-flat }
    \set Staff.shortInstrumentName = \markup { Cl. in B-flat }
    <c' e' g'>4
    d'4
    r4
    e'4
}
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Clarinet in B-flat }
    \set Staff.shortInstrumentName = \markup { Cl. in B-flat }
    <d' fs' a'>4
    e'4
    r4
    fs'4
}
```

Return none.

CHAPTER

NINE

IOTOOLS

9.1 Functions

9.1.1 iotools.clear_terminal

`iotools.clear_terminal()`

New in version 2.0. Run `clear` if OS is POSIX-compliant (UNIX / Linux / MacOS).

Run `cls` if OS is not POSIX-compliant (Windows):

```
>>> iotools.clear_terminal()
```

Return none.

9.1.2 iotools.count_function_calls

`iotools.count_function_calls(`*expr*, *global_context=None*, *local_context=None*, *fixed_point=True*`)`

New in version 2.12. Count function calls returned by `iotools.profile_expr(expr)`.

Example 1. Function calls required to initialize note from string:

```
>>> iotools.count_function_calls("Note('c4')", globals())
11267
```

Example 2. Function calls required to initialize note from integers:

```
>>> iotools.count_function_calls("Note(-12, (1, 4))", globals())
138
```

Return integer.

9.1.3 iotools.f

`iotools.f(`*expr*`)`

Format *expr* and print to standard out:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

Return none.

9.1.4 iotools.get_last_output_file_name

iotools.**get_last_output_file_name**(*path=None*)

Get last output file name like `6222.ly`.

Read Abajd output directory when *path* is none.

Return string.

9.1.5 iotools.get_next_output_file_name

iotools.**get_next_output_file_name**(*file_extension='ly'*, *path=None*)

Get next output file name like `6223.ly`.

Read Abjad output directory when *path* is none.

Return string.

9.1.6 iotools.graph

iotools.**graph**(*expr*, *image_format='pdf'*, *layout='dot'*)

Graph *expr* with graphviz, and open resulting image in the default image viewer:

```
>>> rtm_syntax = '(3 ((2 (2 1)) 2))'
>>> rhythm_tree = rhythmtreetools.RhythmTreeParser()(rtm_syntax)[0]
>>> print rhythm_tree.pretty_rtm_format
(3 (
    (2 (
        2
        1))
    2))
```

```
>>> graph(rhythm_tree)
```

Return None.

9.1.7 iotools.log

iotools.**log**()

Open the LilyPond log file in operating system-specific text editor:

```
>>> iotools.log()
```

```
GNU LilyPond 2.12.2
Processing `0440.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `0440.ps'...
Converting to `./0440.pdf'...
```

Exit text editor in the usual way.

Return none.

9.1.8 iotools.ly

`iotools.ly(target=-1)`

Open the last LilyPond output file in text editor:

```
>>> iotools.ly()
```

```
% Abjad revision 2162
% 2009-05-31 14:29

\version "2.12.2"
\include "english.ly"
\include "/Path/to/abjad/trunk/abjad/cfg/abjad.scm"

{
    c'4
}
```

Open the next-to-last LilyPond output file in text editor:

```
>>> iotools.ly(-2)
```

Exit text editor in the usual way.

Return none.

9.1.9 iotools.p

`iotools.p(arg, language='english')`

New in version 2.7. Parse *arg* as LilyPond string:

```
>>> p("{c'4 d'4 e'4 f'4}")
{c'4, d'4, e'4, f'4}
```

```
>>> container = _
>>> f(container)
{
    c'4
    d'4
    e'4
    f'4
}
```

A pitch-name language may also be specified.

```
>>> p("{c'8 des' e' fis'}", language='nederlands')
{c'8, df'8, e'8, fs'8}
```

Return Abjad expression.

9.1.10 iotools.pdf

`iotools.pdf(target=-1)`

Open the last PDF generated by Abjad with `iotools.pdf()`.

Open the next-to-last PDF generated by Abjad with `iotools.pdf(-2)`.

Return none.

Abjad writes PDFs to the `~/.abjad/output` directory by default.

You may change this by setting the `abjad_output` variable in the `config.py` file.

9.1.11 iotools.play

`iotools.play(expr)`

Play *expr*:

```
>>> note = Note("c'4")
```

```
>>> iotools.play(note)
```

This input creates and opens a one-note MIDI file.

Abjad outputs MIDI files of the format `filename.mid` under Windows.

Abjad outputs MIDI files of the format `filename.midi` under other operating systems.

9.1.12 iotools.plot

`iotools.plot(expr, image_format='png', width=640, height=320)`

Plot *expr* with gnuplot, and open resulting image in the default image viewer.

Return None.

9.1.13 iotools.profile_expr

`iotools.profile_expr(expr, sort_by='cum', line_count=12, strip_dirs=True, print_callers=False, print_callees=False, global_context=None, local_context=None, print_to_terminal=True)`

Profile *expr*:

```
>>> iotools.profile_expr('Staff(notetools.make_repeated_notes(8))')
Tue Apr  5 20:32:40 2011    _tmp_abj_profile

         2852 function calls (2829 primitive calls) in 0.006 CPU seconds

   Ordered by: cumulative time
   List reduced from 118 to 12 due to restriction <12>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.006    0.006 <string>:1(<module>)
        1    0.000    0.000    0.003    0.003 make_repeated_notes.py:5(make_repeated
        1    0.001    0.001    0.003    0.003 make_notes.py:12(make_notes)
        1    0.000    0.000    0.003    0.003 Staff.py:21(__init__)
        1    0.000    0.000    0.003    0.003 Context.py:11(__init__)
        1    0.000    0.000    0.003    0.003 Container.py:23(__init__)
        1    0.000    0.000    0.003    0.003 Container.py:271(_initialize_music)
        2    0.000    0.000    0.002    0.001 all_are_thread_contiguous_components.p
       52    0.001    0.000    0.002    0.000 component_to_thread_signature.py:5(com
        1    0.000    0.000    0.002    0.002 _construct_unprolated_notes.py:4(_cons
        8    0.000    0.000    0.002    0.000 make_tied_note.py:5(make_tied_note)
        8    0.000    0.000    0.002    0.000 make_tied_leaf.py:5(make_tied_leaf)
```

Function wraps the built-in Python `cProfile` module.

Set *expr* to any string of Abjad input.

Set *sort_by* to *'cum'*, *'time'* or *'calls'*.

Set *line_count* to any nonnegative integer.

Set *strip_dirs* to true to strip directory names from output lines.

Function creates the file `_tmp_abj_profile` in the directory from which it is run.

See the Python docs for more information on the Python profilers.

9.1.14 iotools.redo

`iotools.redo(target=-1, lily_time=10)`

Rerender the last `.ly` file created in Abjad and then show the resulting PDF:

```
>>> iotools.redo()
```

Rerender the next-to-last `.ly` file created in Abjad and then show the resulting PDF:

```
>>> iotools.redo(-2)
```

Return none.

9.1.15 iotools.save_last_ly_as

`iotools.save_last_ly_as(file_name)`

New in version 2.0. Save last ly file as *file_name*:

```
>>> iotools.save_last_ly_as('/project/output/example-1.ly')
```

Return none.

9.1.16 iotools.save_last_pdf_as

`iotools.save_last_pdf_as(file_name)`

New in version 2.0. Save last PDF as *file_name*:

```
>>> iotools.save_last_pdf_as('/project/output/example-1.pdf')
```

Return none.

9.1.17 iotools.show

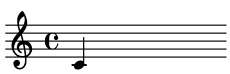
`iotools.show(expr, return_timing=False, suppress_pdf=False, docs=False)`

Show *expr*:

```
>>> note = Note("c'4")
>>> show(note)
```

Show *expr* and return both Abjad and LilyPond processing time in seconds:

```
>>> staff = Staff(Note("c'4") * 200)
>>> show(note, return_timing=True)
(0, 3)
```



Wrap *expr* in a LilyPond file with settings and overrides suitable for the Abjad reference manual When *docs* is true.

Return none or timing tuple.

Abjad writes LilyPond input files to the `~/.abjad/output` directory by default.

You may change this by setting the `abjad_output` variable in the `config.py` file.

9.1.18 iotools.spawn_subprocess

iotools.**spawn_subprocess**(*command*)

New in version 2.9. Spawn subprocess, run *command*, redirect stderr to stdout and print result:

```
>>> iotools.spawn_subprocess('echo "hello world"')
hello world
```

The function is basically a reimplementation of the deprecated `os.system()` using Python's `subprocess` module.

The function provides a type of shell access from the Abjad interpreter.

Return none.

9.1.19 iotools.which

iotools.**which**(*name*, *flags=1*)

Find executable `name`, similar to Unix's `which` command:

```
>>> iotools.which('python2.7')
['/usr/bin/python2.7']
```

Return list of zero or more full paths to `name`.

9.1.20 iotools.write_expr_to_ly

iotools.**write_expr_to_ly**(*expr*, *file_name*, *print_status=False*, *tagline=False*, *docs=False*)

Write *expr* to *file_name*:

```
>>> note = Note("c'4")
>>> iotools.write_expr_to_ly(note, '/home/user/foo.ly')
```

Return none.

9.1.21 iotools.write_expr_to_pdf

iotools.**write_expr_to_pdf**(*expr*, *file_name*, *print_status=False*, *tagline=False*)

Write *expr* to pdf *file_name*:

```
>>> note = Note("c'4")
>>> iotools.write_expr_to_pdf(note, 'one_note.pdf')
```

Return none.

9.1.22 iotools.z

iotools.**z**(*expr*)

New in version 2.8. Print the tools package-qualified indented repr of *z*.

ITERATIONTOOLS

10.1 Functions

10.1.1 iterationtools.iterate_chords_in_expr

iterationtools.**iterate_chords_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate chords forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> f(staff)
\new Staff {
    <e' g' c''>8
    a'8
    r8
    <d' f' b'>8
    r2
}
```

```
>>> for chord in iterationtools.iterate_chords_in_expr(staff):
...   chord
Chord("<e' g' c''>8")
Chord("<d' f' b'>8")
```

Iterate chords backward in *expr*:

```
::
```

```
>>> for chord in iterationtools.iterate_chords_in_expr(staff, reverse=True):
...   chord
Chord("<d' f' b'>8")
Chord("<e' g' c''>8")
```

Ignore threads.

Return generator.

10.1.2 iterationtools.iterate_components_and_grace_containers_in_expr

iterationtools.**iterate_components_and_grace_containers_in_expr**(*expr*, *klass*)

Iterate components of *klass* forward in *expr*:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(voice[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> grace_notes = [Note("c'16"), Note("d'16")]
>>> gracetools.GraceContainer(grace_notes, kind='grace')(voice[1])
Note("d'8")
```

```
>>> after_grace_notes = [Note("e'16"), Note("f'16")]
>>> gracetools.GraceContainer(after_grace_notes, kind='after')(voice[1])
Note("d'8")
```

```
>>> f(voice)
\new Voice {
    c'8 [
    \grace {
        c'16
        d'16
    }
    \afterGrace
    d'8
    {
        e'16
        f'16
    }
    e'8
    f'8 ]
}
```

```
>>> x = iterationtools.iterate_components_and_grace_containers_in_expr(voice, Note)
>>> for note in x:
...     note
...
Note("c'8")
Note("c'16")
Note("d'16")
Note("d'8")
Note("e'16")
Note("f'16")
Note("e'8")
Note("f'8")
```

Include grace leaves before main leaves.

Include grace leaves after main leaves.

10.1.3 iterationtools.iterate_components_depth_first

`iterationtools.iterate_components_depth_first`(*component*, *capped=True*, *unique=True*, *forbid=None*, *direction=Left*)

New in version 1.1. Iterate components depth-first from *component*.

Todo

Add usage examples.

10.1.4 iterationtools.iterate_components_in_expr

`iterationtools.iterate_components_in_expr`(*expr*, *klass=None*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate components forward in *expr*.

10.1.5 iterationtools.iterate_containers_in_expr

`iterationtools.iterate_containers_in_expr`(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate containers forward in *expr*:

```
>>> staff = Staff([Voice("c'8 d'8"), Voice("e'8 f'8 g'8")])
>>> Tuplet(Fraction(2, 3), staff[1][:])
Tuplet(2/3, [e'8, f'8, g'8])
>>> staff.is_parallel = True
```

```
>>> f(staff)
\new Staff <<
    \new Voice {
        c'8
        d'8
    }
    \new Voice {
        \times 2/3 {
            e'8
            f'8
            g'8
        }
    }
>>
```

```
>>> for x in iterationtools.iterate_containers_in_expr(staff):
...     x
Staff<<2>>
Voice{2}
Voice{1}
Tuplet(2/3, [e'8, f'8, g'8])
```

Iterate containers backward in *expr*:

```
>>> for x in iterationtools.iterate_containers_in_expr(staff, reverse=True):
...     x
Staff<<2>>
Voice{1}
Tuplet(2/3, [e'8, f'8, g'8])
Voice{2}
```

Ignore threads.

Return generator.

10.1.6 iterationtools.iterate_contexts_in_expr

`iterationtools.`**`iterate_contexts_in_expr`**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate contexts forward in *expr*:

```
>>> staff = Staff([Voice("c'8 d'8"), Voice("e'8 f'8 g'8")])
>>> Tuplet(Fraction(2, 3), staff[1][:])
Tuplet(2/3, [e'8, f'8, g'8])
>>> staff.is_parallel = True
```

```
>>> f(staff)
\new Staff <<
    \new Voice {
        c'8
        d'8
    }
    \new Voice {
        \times 2/3 {
            e'8
            f'8
            g'8
        }
    }
>>
```

```
>>> for x in iterationtools.iterate_contexts_in_expr(staff):
...     x
Staff<<2>>
```

```
Voice{2}
Voice{1}
```

Iterate contexts backward in *expr*:

```
>>> for x in iterationtools.iterate_contexts_in_expr(staff, reverse=True):
...     x
Staff<<2>>
Voice{1}
Voice{2}
```

Ignore threads.

Return generator.

10.1.7 iterationtools.iterate_leaf_pairs_in_expr

iterationtools.**iterate_leaf_pairs_in_expr**(*expr*)

New in version 2.0. Iterate leaf pairs forward in *expr*:

```
>>> score = Score([])
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'4")]
>>> score.append(Staff(notes))
>>> notes = [Note(x, (1, 4)) for x in [-12, -15, -17]]
>>> score.append(Staff(notes))
>>> contexttools.ClefMark('bass')(score[1])
ClefMark('bass')(Staff{3})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
        g'4
    }
    \new Staff {
        \clef "bass"
        c4
        a,4
        g,4
    }
>>
```

```
>>> for pair in iterationtools.iterate_leaf_pairs_in_expr(score):
...         pair
(Note("c'8"), Note('c4'))
(Note("c'8"), Note("d'8"))
(Note('c4'), Note("d'8"))
(Note("d'8"), Note("e'8"))
(Note("d'8"), Note('a,4'))
(Note('c4'), Note("e'8"))
(Note('c4'), Note('a,4'))
(Note("e'8"), Note('a,4'))
(Note("e'8"), Note("f'8"))
(Note('a,4'), Note("f'8"))
(Note("f'8"), Note("g'4"))
(Note("f'8"), Note('g,4'))
(Note('a,4'), Note("g'4"))
(Note('a,4'), Note('g,4'))
(Note("g'4"), Note('g,4'))
```

Iterate leaf pairs left-to-right and top-to-bottom.

Return generator.

10.1.8 iterationtools.iterate_leaves_in_expr

`iterationtools.iterate_leaves_in_expr(expr, reverse=False, start=0, stop=None)`

New in version 2.10. Iterate leaves forward in *expr*:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 |")
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff):
...     leaf
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("f'8")
Note("g'8")
Note("a'8")
```

Use the optional *start* and *stop* keyword parameters to control the start and stop indices of iteration.

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=3):
...     leaf
...
Note("f'8")
Note("g'8")
Note("a'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=0, stop=3):
...     leaf
...
Note("c'8")
Note("d'8")
Note("e'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, start=2, stop=4):
...     leaf
...
Note("e'8")
Note("f'8")
```

Iterate leaves backward in *expr*:

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(staff, reverse=True):
...     leaf
...
Note("a'8")
Note("g'8")
Note("f'8")
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=3, reverse=True):
...     leaf
...
```

```
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=0, stop=3, reverse=True):
...     leaf
...
Note("a'8")
Note("g'8")
Note("f'8")
```

```
>>> for leaf in iterationtools.iterate_leaves_in_expr(
...     staff, start=2, stop=4, reverse=True):
...     leaf
...
Note("f'8")
Note("e'8")
```

Ignore threads.

Return generator.

10.1.9 iterationtools.iterate_measures_in_expr

`iterationtools.iterate_measures_in_expr`(*expr*, *reverse=False*, *start=0*, *stop=None*)
New in version 2.10. Iterate measures forward in *expr*:

```
>>> staff = Staff("abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 || 2/8 g'8 a'8 |")
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(staff):
...     measure
...
Measure(2/8, [c'8, d'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [g'8, a'8])
```

Use the optional *start* and *stop* keyword parameters to control the start and stop indices of iteration.

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=1):
...     measure
...
Measure(2/8, [e'8, f'8])
Measure(2/8, [g'8, a'8])
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=0, stop=2):
...     measure
...
Measure(2/8, [c'8, d'8])
Measure(2/8, [e'8, f'8])
```

Iterate measures backward in *expr*:

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, reverse=True):
...     measure
...
Measure(2/8, [g'8, a'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [c'8, d'8])
```

Use the optional *start* and *stop* keyword parameters to control indices of iteration.

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=1, reverse=True):
...     measure
...
Measure(2/8, [e'8, f'8])
Measure(2/8, [c'8, d'8])
```

```
>>> for measure in iterationtools.iterate_measures_in_expr(
...     staff, start=0, stop=2, reverse=True):
...     measure
...
Measure(2/8, [g'8, a'8])
Measure(2/8, [e'8, f'8])
```

Ignore threads.

Return generator.

10.1.10 iterationtools.iterate_namesakes_from_component

iterationtools.**iterate_namesakes_from_component**(*component*, *reverse=False*, *start=0*, *stop=None*)

New in version 1.1. Iterate namesakes forward from *component*:

```
>>> container = Container(Staff(notetools.make_repeated_notes(2)) * 2)
>>> container.is_parallel = True
>>> container[0].name = 'staff 1'
>>> container[1].name = 'staff 2'
>>> score = Score([])
>>> score.is_parallel = False
>>> score.extend(container * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(score)
```

```
>>> f(score)
\new Score {
    <<
        \context Staff = "staff 1" {
            c'8
            d'8
        }
        \context Staff = "staff 2" {
            e'8
            f'8
        }
    >>
    <<
        \context Staff = "staff 1" {
            g'8
            a'8
        }
        \context Staff = "staff 2" {
            b'8
            c''8
        }
    >>
}
```

```
>>> for staff in iterationtools.iterate_namesakes_from_component(score[0][0]):
...     print staff.lilypond_format
...
\context Staff = "staff 1" {
    c'8
    d'8
}
\context Staff = "staff 1" {
    g'8
    a'8
}
```

Iterate namesakes backward from *component*:

```
::
```

```
>>> for staff in iterationtools.iterate_namesakes_from_component(
...     score[-1][0], reverse=True):
...     print staff.lilypond_format
...
\context Staff = "staff 1" {
    g'8
    a'8
}
\context Staff = "staff 1" {
    c'8
    d'8
}
```

Return generator.

10.1.11 iterationtools.iterate_notes_and_chords_in_expr

`iterationtools.`**`iterate_notes_and_chords_in_expr`**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate notes and chords forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> f(staff)
\new Staff {
    <e' g' c''>8
    a'8
    r8
    <d' f' b'>8
    r2
}
```

```
>>> for leaf in iterationtools.iterate_notes_and_chords_in_expr(staff):
...   leaf
Chord("<e' g' c''>8")
Note("a'8")
Chord("<d' f' b'>8")
```

Iterate notes and chords backward in *expr*:

```
>>> for leaf in iterationtools.iterate_notes_and_chords_in_expr(staff, reverse=True):
...   leaf
Chord("<d' f' b'>8")
Note("a'8")
Chord("<e' g' c''>8")
```

Ignore threads.

Return generator.

10.1.12 iterationtools.iterate_notes_in_expr

iterationtools.**iterate_notes_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Yield left-to-right notes in *expr*:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("f'8")
Note("g'8")
Note("a'8")
```

Use optional *start* and *stop* keyword parameters to control start and stop indices of iteration:

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, start=3):
...     note
...
Note("f'8")
Note("g'8")
Note("a'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, start=0, stop=3):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, start=2, stop=4):
...     note
...
Note("e'8")
Note("f'8")
```

Yield right-to-left notes in *expr*:

```
>>> staff = Staff()
>>> staff.append(Measure((2, 8), "c'8 d'8"))
>>> staff.append(Measure((2, 8), "e'8 f'8"))
>>> staff.append(Measure((2, 8), "g'8 a'8"))
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
```

```
            c'8
            d'8
        }
        {
            e'8
            f'8
        }
        {
            g'8
            a'8
        }
}
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, reverse=True):
...     note
...
Note("a'8")
Note("g'8")
Note("f'8")
Note("e'8")
Note("d'8")
Note("c'8")
```

Use optional *start* and *stop* keyword parameters to control indices of iteration:

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, reverse=True, start=3):
...     note
...
Note("e'8")
Note("d'8")
Note("c'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, reverse=True, start=0, stop=3):
...     note
...
Note("a'8")
Note("g'8")
Note("f'8")
```

```
>>> for note in iterationtools.iterate_notes_in_expr(staff, reverse=True, start=2, stop=4):
...     note
...
Note("f'8")
Note("e'8")
```

Ignore threads.

Return generator.

10.1.13 iterationtools.iterate_rests_in_expr

`iterationtools.`**`iterate_rests_in_expr`**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate rests forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 r8 <d' f' b'>8 r2")
```

```
>>> f(staff)
\new Staff {
    <e' g' c''>8
    a'8
    r8
    <d' f' b'>8
    r2
}
```

```
>>> for rest in iterationtools.iterate_rests_in_expr(staff):
...   rest
Rest('r8')
Rest('r2')
```

Iterate rests backward in *expr*:

```
>>> for rest in iterationtools.iterate_rests_in_expr(staff, reverse=True):
...     rest
Rest('r2')
Rest('r8')
```

Ignore threads.

Return generator.

10.1.14 iterationtools.iterate_scores_in_expr

iterationtools.**iterate_scores_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate scores forward in *expr*:

```
>>> score_1 = Score([Staff("c'8 d'8 e'8 f'8")])
>>> score_2 = Score([Staff("c'1"), Staff("g'1")])
>>> scores = [score_1, score_2]
```

```
>>> for score in iterationtools.iterate_scores_in_expr(scores):
...     score
Score<<1>>
Score<<2>>
```

Iterate scores backward in *expr*:

```
::
```

```
>>> for score in iterationtools.iterate_scores_in_expr(scores, reverse=True):
...     score
Score<<2>>
Score<<1>>
```

Ignore threads.

Return generator.

10.1.15 iterationtools.iterate_semantic_voices_in_expr

iterationtools.**iterate_semantic_voices_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.0. Iterate semantic voices forward in *expr*:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(3, 8), (5, 16), (5, 16)])
>>> time_signature_voice = Voice(measures)
>>> time_signature_voice.name = 'TimeSignatuerVoice'
>>> time_signature_voice.is_nonsemantic = True
>>> music_voice = Voice("c'4. d'4 e'16 f'4 g'16")
>>> music_voice.name = 'MusicVoice'
>>> staff = Staff([time_signature_voice, music_voice])
>>> staff.is_parallel = True
```

```
>>> f(staff)
\new Staff <<
    \context Voice = "TimeSignatuerVoice" {
        {
            \time 3/8
            s1 * 3/8
        }
        {
            \time 5/16
            s1 * 5/16
        }
        {
            s1 * 5/16
```

```
            }
        }
        \context Voice = "MusicVoice" {
            c'4.
            d'4
            e'16
            f'4
            g'16
        }
    >>

>>> for voice in iterationtools.iterate_semantic_voices_in_expr(staff):
...     voice
Voice-"MusicVoice"{5}
```

Iterate semantic voices backward in *expr*:

```
>>> for voice in iterationtools.iterate_semantic_voices_in_expr(staff, reverse=True):
...     voice
Voice-"MusicVoice"{5}
```

Return generator.

10.1.16 iterationtools.iterate_skips_in_expr

iterationtools.**iterate_skips_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate skips forward in *expr*:

```
>>> staff = Staff("<e' g' c''>8 a'8 s8 <d' f' b'>8 s2")
```

```
>>> f(staff)
\new Staff {
    <e' g' c''>8
    a'8
    s8
    <d' f' b'>8
    s2
}
```

```
>>> for skip in iterationtools.iterate_skips_in_expr(staff):
...     skip
Skip('s8')
Skip('s2')
```

Iterate skips backwards in *expr*:

```
>>> for skip in iterationtools.iterate_skips_in_expr(staff, reverse=True):
...     skip
Skip('s2')
Skip('s8')
```

Ignore threads.

Return generator.

10.1.17 iterationtools.iterate_staves_in_expr

iterationtools.**iterate_staves_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate staves forward in *expr*:

```
>>> score = Score(4 * Staff([]))
```

```
>>> f(score)
\new Score <<
    \new Staff {
    }
```

```
        \new Staff {
        }
        \new Staff {
        }
        \new Staff {
        }
    >>
```

```
>>> for staff in iterationtools.iterate_staves_in_expr(score):
...     staff
...
Staff{}
Staff{}
Staff{}
Staff{}
```

Iterate staves backward in *expr*:

```
::
```

```
>>> for staff in iterationtools.iterate_staves_in_expr(score, reverse=True):
...     staff
...
Staff{}
Staff{}
Staff{}
Staff{}
```

Return generator.

10.1.18 iterationtools.iterate_thread_from_component

`iterationtools.iterate_thread_from_component(component, klass=None, reverse=False)`

New in version 2.10. Itearte thread forward from *component* and yield instances of *klass*:

```
>>> container = Container(Voice(notetools.make_repeated_notes(2)) * 2)
>>> container.is_parallel = True
>>> container[0].name = 'voice 1'
>>> container[1].name = 'voice 2'
>>> staff = Staff(container * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    <<
        \context Voice = "voice 1" {
            c'8
            d'8
        }
        \context Voice = "voice 2" {
            e'8
            f'8
        }
    >>
    <<
        \context Voice = "voice 1" {
            g'8
            a'8
        }
        \context Voice = "voice 2" {
            b'8
            c''8
        }
    >>
}
```

Starting from the first leaf in score.

```
>>> for x in iterationtools.iterate_thread_from_component(
...     staff.leaves[0], Note):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Starting from the second leaf in score.

```
>>> for x in iterationtools.iterate_thread_from_component(
...     staff.leaves[1], Note):
...     x
...
Note("d'8")
Note("g'8")
Note("a'8")
```

Yield all components in thread.

```
>>> for x in iterationtools.iterate_thread_from_component(
...     staff.leaves[0]):
...     x
...
Note("c'8")
Voice-"voice 1"{2}
Note("d'8")
Voice-"voice 1"{2}
Note("g'8")
Note("a'8")
```

Iterate thread backward from *component* and yield instances of *klass*, starting from the last leaf in score.

```
>>> for x in iterationtools.iterate_thread_from_component(
...     staff.leaves[-1], Note, reverse=True):
...     x
Note("c''8")
Note("b'8")
Note("f'8")
Note("e'8")
```

Yield all components in thread:

```
>>> for x in iterationtools.iterate_thread_from_component(
...     staff.leaves[-1], reverse=True):
...     x
Note("c''8")
Voice-"voice 2"{2}
Note("b'8")
Voice-"voice 2"{2}
Note("f'8")
Note("e'8")
```

Return generator.

10.1.19 iterationtools.iterate_thread_in_expr

`iterationtools.`**`iterate_thread_in_expr`**(*expr*, *klass*, *containment_signature*, *reverse=False*)

New in version 2.10. Yield left-to-right instances of *klass* in *expr* with *containment_signature*:

```
>>> container = Container(Voice(notetools.make_repeated_notes(2)) * 2)
>>> container.is_parallel = True
>>> container[0].name = 'voice 1'
>>> container[1].name = 'voice 2'
>>> staff = Staff(container * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    <<
        \context Voice = "voice 1" {
            c'8
            d'8
        }
        \context Voice = "voice 2" {
            e'8
            f'8
        }
    >>
    <<
        \context Voice = "voice 1" {
            g'8
            a'8
        }
        \context Voice = "voice 2" {
            b'8
            c''8
        }
    >>
}
```

```
>>> signature = staff.leaves[0].parentage.containment_signature
>>> for x in iterationtools.iterate_thread_in_expr(staff, Note, signature):
...     x
...
Note("c'8")
Note("d'8")
Note("g'8")
Note("a'8")
```

Return generator.

10.1.20 iterationtools.iterate_timeline_from_component

iterationtools.**iterate_timeline_from_component**(*expr*, *klass=None*, *reverse=False*)
New in version 2.10. Iterate timeline forward from *component*:

```
>>> score = Score([])
>>> score.append(Staff(notetools.make_repeated_notes(4, Duration(1, 4))))
>>> score.append(Staff(notetools.make_repeated_notes(4)))
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
    \new Staff {
        g'8
        a'8
        b'8
        c''8
    }
>>
```

```
>>> for leaf in iterationtools.iterate_timeline_from_component(score[1][2]):
...     leaf
...
Note("b'8")
Note("c''8")
Note("e'4")
Note("f'4")
```

Iterate timeline backward from *component*:

```
::
```

```
>>> for leaf in iterationtools.iterate_timeline_from_component(score[1][2], reverse=True):
...     leaf
...
Note("b'8")
Note("c'4")
Note("a'8")
Note("g'8")
```

Yield components sorted backward by score offset stop time when *reverse* is True.

Iterate leaves when *klass* is none.

Todo

optimize to avoid behind-the-scenes full-score traversal.

10.1.21 iterationtools.iterate_timeline_in_expr

iterationtools.**iterate_timeline_in_expr**(*expr*, *klass=None*, *reverse=False*)
New in version 2.10. Iterate timeline forward in *expr*:

```
>>> score = Score([])
>>> score.append(Staff(notetools.make_repeated_notes(4, Duration(1, 4))))
>>> score.append(Staff(notetools.make_repeated_notes(4)))
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
    \new Staff {
        g'8
        a'8
        b'8
        c''8
    }
>>
```

```
>>> for leaf in iterationtools.iterate_timeline_in_expr(score):
...     leaf
...
Note("c'4")
Note("g'8")
Note("a'8")
Note("d'4")
Note("b'8")
Note("c''8")
Note("e'4")
Note("f'4")
```

Iterate timeline backward in *expr*:

```
::
```

```
>>> for leaf in iterationtools.iterate_timeline_in_expr(score, reverse=True):
...     leaf
...
Note("f'4")
Note("e'4")
Note("d'4")
```

```
Note("c''8")
Note("b'8")
Note("c'4")
Note("a'8")
Note("g'8")
```

Iterate leaves when *klass* is none.

Todo

optimize to avoid behind-the-scenes full-score traversal.

10.1.22 iterationtools.iterate_tuplets_in_expr

iterationtools.**iterate_tuplets_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.10. Iterate tuplets forward in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> Tuplet(Fraction(2, 3), staff[:3])
Tuplet(2/3, [c'8, d'8, e'8])
>>> Tuplet(Fraction(2, 3), staff[-3:])
Tuplet(2/3, [a'8, b'8, c''8])

>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    f'8
    g'8
    \times 2/3 {
        a'8
        b'8
        c''8
    }
}

>>> for tuplet in iterationtools.iterate_tuplets_in_expr(staff):
...     tuplet
...
Tuplet(2/3, [c'8, d'8, e'8])
Tuplet(2/3, [a'8, b'8, c''8])
```

Iterate tuplets backward in *expr*:

```
>>> for tuplet in iterationtools.iterate_tuplets_in_expr(staff, reverse=True):
...     tuplet
...
Tuplet(2/3, [a'8, b'8, c''8])
Tuplet(2/3, [c'8, d'8, e'8])
```

Ignore threads.

Return generator.

10.1.23 iterationtools.iterate_voices_in_expr

iterationtools.**iterate_voices_in_expr**(*expr*, *reverse=False*, *start=0*, *stop=None*)

New in version 2.0. Iterate voices forward in *expr*:

```
>>> voice_1 = Voice("c'8 d'8 e'8 f'8")
>>> voice_2 = Voice("c'4 b4")
>>> staff = Staff([voice_1, voice_2])
>>> staff.is_parallel = True
```

```
>>> f(staff)
\new Staff <<
    \new Voice {
        c'8
        d'8
        e'8
        f'8
    }
    \new Voice {
        c'4
        b4
    }
>>
```

```
>>> for voice in iterationtools.iterate_voices_in_expr(staff):
...     voice
Voice{4}
Voice{2}
```

Iterate voices backward in *expr*:

```
::
```

```
>>> for voice in iterationtools.iterate_voices_in_expr(staff, reverse=True):
...     voice
Voice{2}
Voice{4}
```

Return generator.

CHAPTER

ELEVEN

LABELTOOLS

11.1 Functions

11.1.1 labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map

labeltools.**color_chord_note_heads_in_expr_by_pitch_class_color_map**(*chord*, *color_map*)

New in version 2.0. Color *chord* note heads by pitch-class *color_map*:

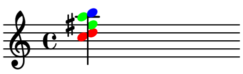
```
>>> chord = Chord([12, 14, 18, 21, 23], (1, 4))
```

```
>>> pitches = [[-12, -10, 4], [-2, 8, 11, 17], [19, 27, 30, 33, 37]]
>>> colors = ['red', 'blue', 'green']
>>> color_map = pitchtools.NumberedChromaticPitchClassColorMap(pitches, colors)
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(chord, color_map)
Chord("<c'' d'' fs'' a'' b''>4")
```

```
>>> f(chord)
<
    \tweak #'color #red
    c''
    \tweak #'color #red
    d''
    \tweak #'color #green
    fs''
    \tweak #'color #green
    a''
    \tweak #'color #blue
    b''
>4
```

```
>>> show(chord)
```



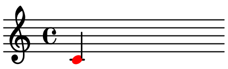
Also works on notes:

```
>>> note = Note("c'4")
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(note, color_map)
Note("c'4")
```

```
>>> f(note)
\once \override NoteHead #'color = #red
c'4
```

```
>>> show(note)
```



When *chord* is neither a chord nor note return *chord* unchanged:

```
>>> staff = Staff([])
```

```
>>> labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map(staff, color_map)
Staff{}
```

Return *chord*.

11.1.2 labeltools.color_contents_of_container

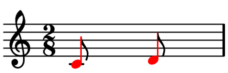
labeltools.**color_contents_of_container**(*container*, *color*)
New in version 2.0. Color contents of *container*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
```

```
>>> labeltools.color_contents_of_container(measure, 'red')
Measure(2/8, [c'8, d'8])
```

```
>>> f(measure)
{
    \override Accidental #'color = #red
    \override Beam #'color = #red
    \override Dots #'color = #red
    \override NoteHead #'color = #red
    \override Rest #'color = #red
    \override Stem #'color = #red
    \override TupletBracket #'color = #red
    \override TupletNumber #'color = #red
    \time 2/8
    c'8
    d'8
    \revert Accidental #'color
    \revert Beam #'color
    \revert Dots #'color
    \revert NoteHead #'color
    \revert Rest #'color
    \revert Stem #'color
    \revert TupletBracket #'color
    \revert TupletNumber #'color
}
```

```
>>> show(measure)
```



Return none.

11.1.3 labeltools.color_leaf

labeltools.**color_leaf**(*leaf*, *color*)
New in version 2.0. Color note:

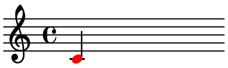
```
>>> note = Note("c'4")
```

```
>>> labeltools.color_leaf(note, 'red')
Note("c'4")
```

```
>>> f(note)
\once \override Accidental #'color = #red
\once \override Dots #'color = #red
```

```
\once \override NoteHead #'color = #red
c'4
```

```
>>> show(note)
```



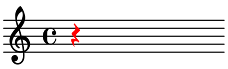
Color rest:

```
>>> rest = Rest('r4')
```

```
>>> labeltools.color_leaf(rest, 'red')
Rest('r4')
```

```
>>> f(rest)
\once \override Dots #'color = #red
\once \override Rest #'color = #red
r4
```

```
>>> show(rest)
```



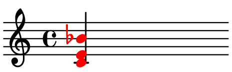
Color chord:

```
>>> chord = Chord("<c' e' bf'>4")
```

```
>>> labeltools.color_leaf(chord, 'red')
Chord("<c' e' bf'>4")
```

```
>>> f(chord)
\once \override Accidental #'color = #red
\once \override Dots #'color = #red
\once \override NoteHead #'color = #red
<c' e' bf'>4
```

```
>>> show(chord)
```



Return *leaf*.

11.1.4 labeltools.color_leaves_in_expr

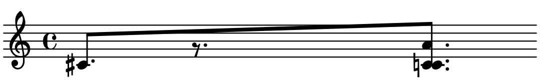
`labeltools.`**`color_leaves_in_expr`**(*expr*, *color*)

New in version 2.0. Color leaves in *expr*:

```
>>> staff = Staff("cs'8. [ r8. s8. <c' cs' a'>8. ]")
```

```
>>> f(staff)
\new Staff {
    cs'8. [
    r8.
    s8.
    <c' cs' a'>8. ]
}
```

```
>>> show(staff)
```



```
>>> labeltools.color_leaves_in_expr(staff, 'red')
```

```
>>> f(staff)
\new Staff {
    \once \override Accidental #'color = #red
    \once \override Dots #'color = #red
    \once \override NoteHead #'color = #red
    cs'8. [
    \once \override Dots #'color = #red
    \once \override Rest #'color = #red
    r8.
    s8.
    \once \override Accidental #'color = #red
    \once \override Dots #'color = #red
    \once \override NoteHead #'color = #red
    <c' cs' a'>8. ]
}
```

```
>>> show(staff)
```

Return none.

11.1.5 labeltools.color_measure

labeltools.**color_measure**(*measure*, *color='red'*)
New in version 2.0. Color *measure* with *color*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
```

```
>>> f(measure)
{
    \time 2/8
    c'8
    d'8
}
```

```
>>> show(measure)
```

```
>>> labeltools.color_measure(measure, 'red')
Measure(2/8, [c'8, d'8])
```

```
>>> f(measure)
{
    \override Beam #'color = #red
    \override Dots #'color = #red
    \override NoteHead #'color = #red
    \override Staff.TimeSignature #'color = #red
    \override Stem #'color = #red
    \time 2/8
    c'8
    d'8
    \revert Beam #'color
    \revert Dots #'color
    \revert NoteHead #'color
    \revert Staff.TimeSignature #'color
    \revert Stem #'color
}
```

```
>>> show(measure)
```

Return colored *measure*.

Color names appear in LilyPond Learning Manual appendix B.5.

11.1.6 labeltools.color_measures_with_non_power_of_two_denominators_in_expr

labeltools.**color_measures_with_non_power_of_two_denominators_in_expr**(*expr*, *color='red'*)

New in version 2.0. Color measures with non-power-of-two denominators in *expr* with *color*:

```
>>> staff = Staff(Measure((2, 8), "c'8 d'8") * 2)
>>> measuretools.scale_measure_denominator_and_adjust_measure_contents(staff[1], 3)
Measure(3/12, [c'8., d'8.])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        \time 3/12
        \scaleDurations #'(2 . 3) {
            c'8.
            d'8.
        }
    }
}
```

```
>>> show(staff)
```

```
>>> labeltools.color_measures_with_non_power_of_two_denominators_in_expr(staff, 'red')
[Measure(3/12, [c'8., d'8.])]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        \override Beam #'color = #red
        \override Dots #'color = #red
        \override NoteHead #'color = #red
        \override Staff.TimeSignature #'color = #red
        \override Stem #'color = #red
        \time 3/12
        \scaleDurations #'(2 . 3) {
            c'8.
            d'8.
        }
        \revert Beam #'color
        \revert Dots #'color
        \revert NoteHead #'color
        \revert Staff.TimeSignature #'color
        \revert Stem #'color
    }
}
```

```
>>> show(staff)
```

Return list of measures colored.

Color names appear in LilyPond Learning Manual appendix B.5.

11.1.7 labeltools.color_note_head_by_numbered_chromatic_pitch_class_color_map

labeltools.**color_note_head_by_numbered_chromatic_pitch_class_color_map**(*pitch_carrier*)

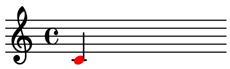
Color *pitch_carrier* note head:

```
>>> note = Note("c'4")
```

```
>>> labeltools.color_note_head_by_numbered_chromatic_pitch_class_color_map(note)
Note("c'4")
```

```
>>> f(note)
\once \override NoteHead #'color = #(x11-color 'red)
c'4
```

```
>>> show(note)
```



Numbered chromatic pitch-class color map:

```
0: red
1: MediumBlue
2: orange
3: LightSlateBlue
4: ForestGreen
5: MediumOrchid
6: firebrick
7: DeepPink
8: DarkOrange
9: IndianRed
10: CadetBlue
11: SeaGreen
12: LimeGreen
```

Numbered chromatic pitch-class color map can not be changed.

Raise type error when *pitch_carrier* is not a pitch carrier.

Raise extra pitch error when *pitch_carrier* carries more than 1 note head.

Raise missing pitch error when *pitch_carrier* carries no note head.

Return *pitch_carrier*.

11.1.8 labeltools.label_leaves_in_expr_with_inversion_equivalent_chromatic_interval_classe

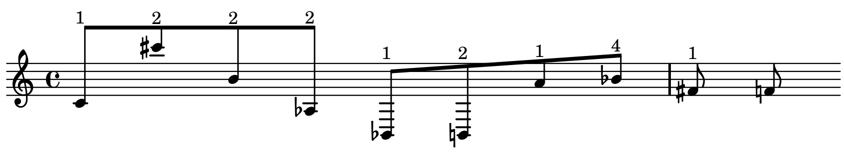
labeltools.**label_leaves_in_expr_with_inversion_equivalent_chromatic_interval_classes**(*expr*
mar

New in version 2.0. Label leaves in *expr* with inversion-equivalent chromatic interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_inversion_equivalent_chromatic_interval_classes(
...     staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { 1 }
    cs'''8 ^ \markup { 2 }
    b'8 ^ \markup { 2 }
    af8 ^ \markup { 2 }
    bf,8 ^ \markup { 1 }
    b,8 ^ \markup { 2 }
    a'8 ^ \markup { 1 }
    bf'8 ^ \markup { 4 }
    fs'8 ^ \markup { 1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.9 labeltools.label_leaves_in_expr_with_leaf_depth

labeltools.**label_leaves_in_expr_with_leaf_depth**(*expr*, *markup_direction=Down*)

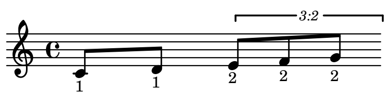
New in version 1.1. Label leaves in *expr* with leaf depth:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> tuplettools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(1/4, [e'8, f'8, g'8])
```

```
>>> labeltools.label_leaves_in_expr_with_leaf_depth(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 1 }
    d'8 _ \markup { \small 1 }
    \times 2/3 {
        e'8 _ \markup { \small 2 }
        f'8 _ \markup { \small 2 }
        g'8 _ \markup { \small 2 }
    }
}
```

```
>>> show(staff)
```



Return none.

11.1.10 labeltools.label_leaves_in_expr_with_leaf_duration

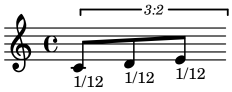
labeltools.**label_leaves_in_expr_with_leaf_duration**(*expr*, *markup_direction=Down*)

New in version 1.1. Label leaves in *expr* with prolated leaf duration:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
>>> labeltools.label_leaves_in_expr_with_leaf_duration(tuplet)
>>> f(tuplet)
\times 2/3 {
    c'8 _ \markup { \small 1/12 }
```

```
        d'8 _ \markup { \small 1/12 }
        e'8 _ \markup { \small 1/12 }
    }
```

```
>>> show(tuplet)
```



Return none.

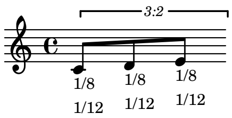
11.1.11 labeltools.label_leaves_in_expr_with_leaf_durations

labeltools.**label_leaves_in_expr_with_leaf_durations**(*expr*, *markup_direction=Down*)

New in version 1.1. Label leaves in *expr* with leaf durations:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
>>> labeltools.label_leaves_in_expr_with_leaf_durations(tuplet)
>>> f(tuplet)
\times 2/3 {
    c'8 _ \markup { \column { \small 1/8 \small 1/12 } }
    d'8 _ \markup { \column { \small 1/8 \small 1/12 } }
    e'8 _ \markup { \column { \small 1/8 \small 1/12 } }
}
```

```
>>> show(tuplet)
```



Label both written duration and prolated duration.

Return none.

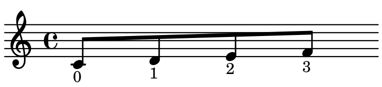
11.1.12 labeltools.label_leaves_in_expr_with_leaf_indices

labeltools.**label_leaves_in_expr_with_leaf_indices**(*expr*, *markup_direction=Down*)

New in version 2.0. Label leaves in *expr* with leaf indices:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_leaf_indices(staff)
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 0 }
    d'8 _ \markup { \small 1 }
    e'8 _ \markup { \small 2 }
    f'8 _ \markup { \small 3 }
}
```

```
>>> show(staff)
```



Return none.

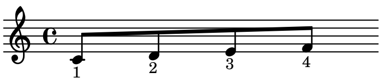
11.1.13 labeltools.label_leaves_in_expr_with_leaf_numbers

labeltools.**label_leaves_in_expr_with_leaf_numbers**(*expr*, *markup_direction=Down*)

New in version 1.1. Label leaves in *expr* with leaf numbers:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_leaf_numbers(staff)
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 1 }
    d'8 _ \markup { \small 2 }
    e'8 _ \markup { \small 3 }
    f'8 _ \markup { \small 4 }
}
```

```
>>> show(staff)
```



Number leaves starting from 1.

Return none.

11.1.14 labeltools.label_leaves_in_expr_with_melodic_chromatic_interval_classes

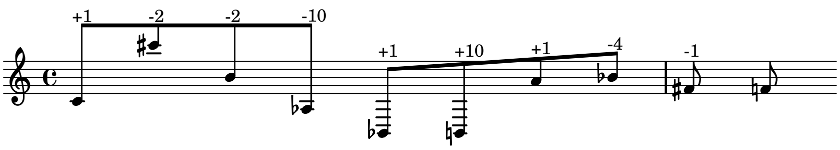
labeltools.**label_leaves_in_expr_with_melodic_chromatic_interval_classes**(*expr*, *markup_direction=Up*)

New in version 2.0. Label leaves in *expr* with melodic chromatic interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_chromatic_interval_classes(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +1 }
    cs'''8 ^ \markup { -2 }
    b'8 ^ \markup { -2 }
    af8 ^ \markup { -10 }
    bf,8 ^ \markup { +1 }
    b,8 ^ \markup { +10 }
    a'8 ^ \markup { +1 }
    bf'8 ^ \markup { -4 }
    fs'8 ^ \markup { -1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.15 labeltools.label_leaves_in_expr_with_melodic_chromatic_intervals

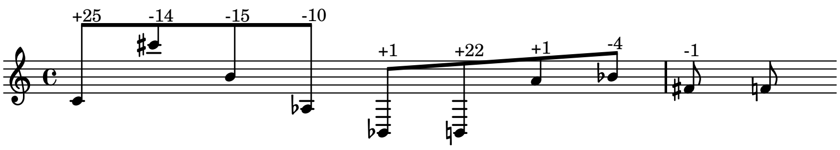
labeltools.**label_leaves_in_expr_with_melodic_chromatic_intervals**(*expr*, *markup_direction=Up*)

New in version 2.0. Label leaves in *expr* with melodic chromatic intervals:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_chromatic_intervals(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +25 }
    cs'''8 ^ \markup { -14 }
    b'8 ^ \markup { -15 }
    af8 ^ \markup { -10 }
    bf,8 ^ \markup { +1 }
    b,8 ^ \markup { +22 }
    a'8 ^ \markup { +1 }
    bf'8 ^ \markup { -4 }
    fs'8 ^ \markup { -1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.16 labeltools.label_leaves_in_expr_with_melodic_counterpoint_interval_classes

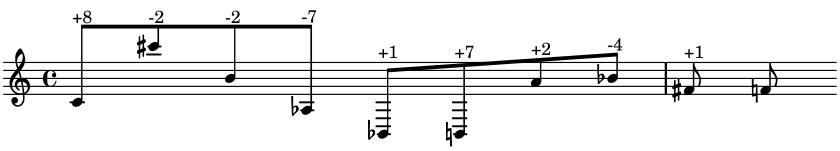
`labeltools.`**`label_leaves_in_expr_with_melodic_counterpoint_interval_classes`**(*expr*, *markup_direction=*

New in version 2.0. Label leaves in *expr* with melodic counterpoint interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_counterpoint_interval_classes(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +8 }
    cs'''8 ^ \markup { -2 }
    b'8 ^ \markup { -2 }
    af8 ^ \markup { -7 }
    bf,8 ^ \markup { +1 }
    b,8 ^ \markup { +7 }
    a'8 ^ \markup { +2 }
    bf'8 ^ \markup { -4 }
    fs'8 ^ \markup { +1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.17 labeltools.label_leaves_in_expr_with_melodic_counterpoint_intervals

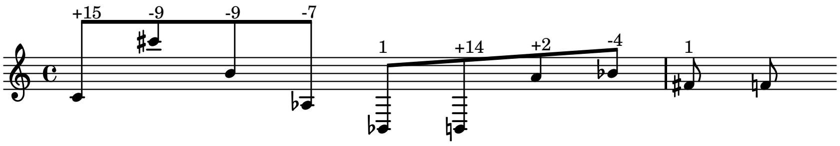
labeltools.**label_leaves_in_expr_with_melodic_counterpoint_intervals**(*expr*, *markup_direction=Up*)

New in version 2.0. Label leaves in *expr* with melodic counterpoint intervals:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_counterpoint_intervals(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +15 }
    cs'''8 ^ \markup { -9 }
    b'8 ^ \markup { -9 }
    af8 ^ \markup { -7 }
    bf,8 ^ \markup { 1 }
    b,8 ^ \markup { +14 }
    a'8 ^ \markup { +2 }
    bf'8 ^ \markup { -4 }
    fs'8 ^ \markup { 1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.18 labeltools.label_leaves_in_expr_with_melodic_diatonic_interval_classes

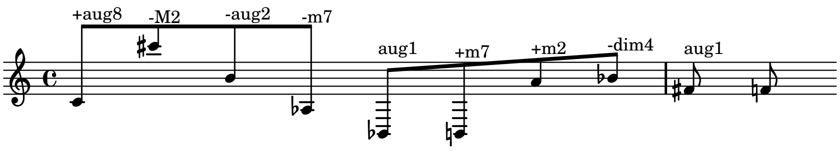
labeltools.**label_leaves_in_expr_with_melodic_diatonic_interval_classes**(*expr*, *markup_direction=Up*)

New in version 2.0. Label leaves in *expr* with melodic diatonic interval classes:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_diatonic_interval_classes(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +aug8 }
    cs'''8 ^ \markup { -M2 }
    b'8 ^ \markup { -aug2 }
    af8 ^ \markup { -m7 }
    bf,8 ^ \markup { aug1 }
    b,8 ^ \markup { +m7 }
    a'8 ^ \markup { +m2 }
    bf'8 ^ \markup { -dim4 }
    fs'8 ^ \markup { aug1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

11.1.19 labeltools.label_leaves_in_expr_with_melodic_diatonic_intervals

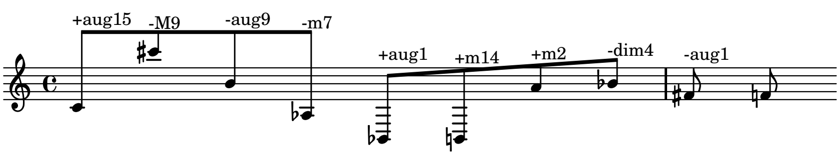
labeltools.**label_leaves_in_expr_with_melodic_diatonic_intervals**(*expr*, *markup_direction=Up*)

New in version 2.0. Label leaves in *expr* with melodic diatonic intervals:

```
>>> notes = notetools.make_notes([0, 25, 11, -4, -14, -13, 9, 10, 6, 5], [Duration(1, 8)])
>>> staff = Staff(notes)
>>> labeltools.label_leaves_in_expr_with_melodic_diatonic_intervals(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ^ \markup { +aug15 }
    cs'''8 ^ \markup { -M9 }
    b'8 ^ \markup { -aug9 }
    af8 ^ \markup { -m7 }
    bf,8 ^ \markup { +aug1 }
    b,8 ^ \markup { +m14 }
    a'8 ^ \markup { +m2 }
    bf'8 ^ \markup { -dim4 }
    fs'8 ^ \markup { -aug1 }
    f'8
}
```

```
>>> show(staff)
```



Return none.

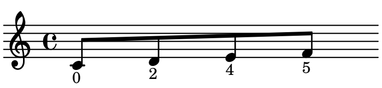
11.1.20 labeltools.label_leaves_in_expr_with_pitch_class_numbers

labeltools.**label_leaves_in_expr_with_pitch_class_numbers**(*expr*, *number=True*, *color=False*, *markup_direction=Down*)

New in version 1.1. Label leaves in *expr* with pitch-class numbers:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(staff)
>>> print staff.lilypond_format
\new Staff {
    c'8 _ \markup { \small 0 }
    d'8 _ \markup { \small 2 }
    e'8 _ \markup { \small 4 }
    f'8 _ \markup { \small 5 }
}
```

```
>>> show(staff)
```



When `color=True` call `color_note_head_by_numbered_chromatic_pitch_class_color_map()`:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(
...     staff, color=True, number=False)
>>> print staff.lilypond_format
\new Staff {
    \once \override NoteHead #'color = #(x11-color 'red)
    c'8
    \once \override NoteHead #'color = #(x11-color 'orange)
    d'8
```

```
        \once \override NoteHead #'color = #(x11-color 'ForestGreen)
        e'8
        \once \override NoteHead #'color = #(x11-color 'MediumOrchid)
        f'8
    }
```

```
>>> show(staff)
```

You can set *number* and *color* at the same time.

Return none.

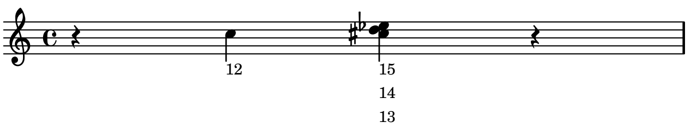
11.1.21 labeltools.label_leaves_in_expr_with_pitch_numbers

`labeltools.`**`label_leaves_in_expr_with_pitch_numbers`**`(`*expr*, *markup_direction=Down*`)`

New in version 1.1. Label leaves in *expr* with pitch numbers:

```
>>> staff = Staff(leaftools.make_leaves([None, 12, [13, 14, 15], None], [(1, 4)]))
>>> labeltools.label_leaves_in_expr_with_pitch_numbers(staff)
>>> f(staff)
\new Staff {
    r4
    c''4 _ \markup { \small 12 }
    <cs'' d'' ef''>4 _ \markup { \column { \small 15 \small 14 \small 13 } }
    r4
}
```

```
>>> show(staff)
```



Return none.

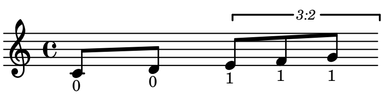
11.1.22 labeltools.label_leaves_in_expr_with_tuplet_depth

`labeltools.`**`label_leaves_in_expr_with_tuplet_depth`**`(`*expr*, *markup_direction=Down*`)`

New in version 1.1. Label leaves in *expr* with tuplet depth:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> tuplettools.FixedDurationTuplet(Duration(2, 8), staff[-3:])
FixedDurationTuplet(1/4, [e'8, f'8, g'8])
>>> labeltools.label_leaves_in_expr_with_tuplet_depth(staff)
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 0 }
    d'8 _ \markup { \small 0 }
    \times 2/3 {
        e'8 _ \markup { \small 1 }
        f'8 _ \markup { \small 1 }
        g'8 _ \markup { \small 1 }
    }
}
```

```
>>> show(staff)
```



Return none.

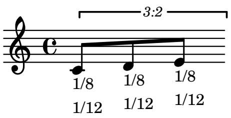
11.1.23 labeltools.label_leaves_in_expr_with_written_leaf_duration

labeltools.**label_leaves_in_expr_with_written_leaf_duration**(*expr*, *markup_direction=Down*)

New in version 1.1. Label leaves in *expr* with writen leaf duration:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
>>> labeltools.label_leaves_in_expr_with_leaf_durations(tuplet)
>>> f(tuplet)
\times 2/3 {
    c'8 _ \markup { \column { \small 1/8 \small 1/12 } }
    d'8 _ \markup { \column { \small 1/8 \small 1/12 } }
    e'8 _ \markup { \column { \small 1/8 \small 1/12 } }
}
```

```
>>> show(tuplet)
```



Return none.

11.1.24 labeltools.label_notes_in_expr_with_note_indices

labeltools.**label_notes_in_expr_with_note_indices**(*expr*, *markup_direction=Down*)

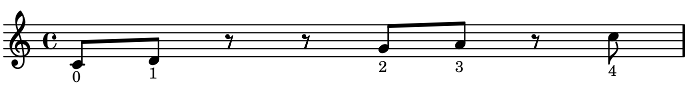
New in version 2.0. Label notes in *expr* with note indices:

```
>>> staff = Staff("c'8 d'8 r8 r8 g'8 a'8 r8 c''8")
```

```
>>> labeltools.label_notes_in_expr_with_note_indices(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 0 }
    d'8 _ \markup { \small 1 }
    r8
    r8
    g'8 _ \markup { \small 2 }
    a'8 _ \markup { \small 3 }
    r8
    c''8 _ \markup { \small 4 }
}
```

```
>>> show(staff)
```



Return none.

11.1.25 labeltools.label_tie_chains_in_expr_with_tie_chain_duration

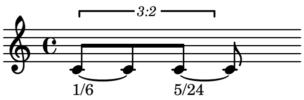
labeltools.**label_tie_chains_in_expr_with_tie_chain_duration**(*expr*, *markup_direction=Down*)

Label tie chains in *expr* with prolated tie chain duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_tie_chain_duration(staff)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 ~
            _ \markup {
                \small
                    1/6
                }
        c'8
        c'8 ~
            _ \markup {
                \small
                    5/24
                }
    }
    c'8
}
```

```
>>> show(staff)
```



Return none.

11.1.26 labeltools.label_tie_chains_in_expr_with_tie_chain_durations

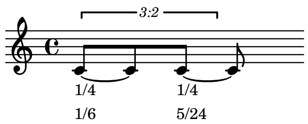
`labeltools.`**`label_tie_chains_in_expr_with_tie_chain_durations`**(*expr*, *markup_direction=Down*)

Label tie chains in *expr* with both written tie chain duration and prolated tie chain duration:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_tie_chain_durations(staff)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 ~
            _ \markup {
                \column
                    {
                        \small
                            1/4
                        \small
                            1/6
                    }
                }
        c'8
        c'8 ~
            _ \markup {
                \column
                    {
                        \small
                            1/4
                        \small
                            5/24
                    }
                }
    }
    c'8
}
```

```
>>> show(staff)
```



Return none.

11.1.27 labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration

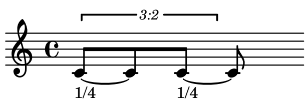
labeltools.**label_tie_chains_in_expr_with_written_tie_chain_duration**(*expr*, *markup_direction=Down*)

Label tie chains in *expr* with written tie chain duration.:

```
>>> staff = Staff(r"\times 2/3 { c'8 ~ c'8 c'8 ~ } c'8")
>>> labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration(staff)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 ~
            _ \markup {
                \small
                    1/4
                }
        c'8
        c'8 ~
            _ \markup {
                \small
                    1/4
                }
    }
    c'8
}
```

```
>>> show(staff)
```



Return none.

11.1.28 labeltools.label_vertical_moments_in_expr_with_chromatic_interval_classes

labeltools.**label_vertical_moments_in_expr_with_chromatic_interval_classes**(*expr*, *markup_direction=I*

New in version 2.0. Label harmonic chromatic interval-classes of every vertical moment in *expr*:

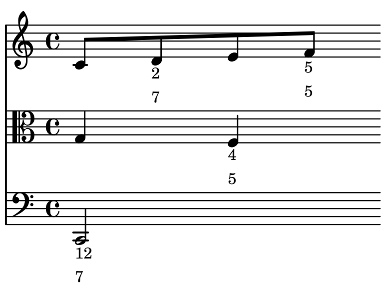
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_chromatic_interval_classes(
...     score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
```

```
                \small
                    \column
                        {
                            2
                            7
                        }
                }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            5
                            5
                        }
                }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \small
                    \column
                        {
                            4
                            5
                        }
                }
    }
    \new Staff {
        \clef "bass"
        c,2
            _ \markup {
                \small
                    \column
                        {
                            12
                            7
                        }
                }
    }
>>
```

```
>>> show(score)
```



Return none.

11.1.29 labeltools.label_vertical_moments_in_expr_with_chromatic_intervals

`labeltools.`**`label_vertical_moments_in_expr_with_chromatic_intervals`**(*expr*, *markup_direction=Down*)

New in version 2.0. Label harmonic chromatic intervals of every vertical moment in *expr*:

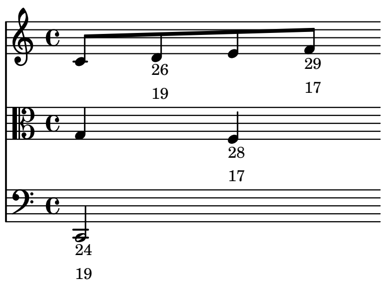
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
```

```
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_chromatic_intervals(
...     score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \small
                    \column
                        {
                            26
                            19
                        }
                }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            29
                            17
                        }
                }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \small
                    \column
                        {
                            28
                            17
                        }
                }
    }
    \new Staff {
        \clef "bass"
        c,2
            _ \markup {
                \small
                    \column
                        {
                            24
                            19
                        }
                }
    }
>>
```

```
>>> show(score)
```



Return none.

11.1.30 labeltools.label_vertical_moments_in_expr_with_counterpoint_intervals

labeltools.**label_vertical_moments_in_expr_with_counterpoint_intervals**(*expr*, *markup_direction=Down*)

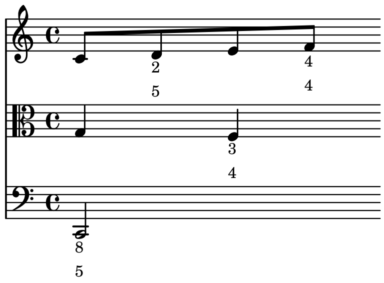
New in version 2.0. Label counterpoint interval of every vertical moment in *expr*:

```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_counterpoint_intervals(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \small
                    \column
                        {
                            2
                            5
                        }
                }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            4
                            4
                        }
                }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \small
                    \column
                        {
                            3
                            4
                        }
                }
    }
    \new Staff {
        \clef "bass"
        c,2
            _ \markup {
                \small
                    \column
                        {
                            8
                            5
                        }
                }
    }
>>
```

```
>>> show(score)
```



Return none.

11.1.31 labeltools.label_vertical_moments_in_expr_with_diatonic_intervals

labeltools.**label_vertical_moments_in_expr_with_diatonic_intervals**(*expr*, *markup_direction=Down*)

New in version 2.0. Label diatonic intervals of every vertical moment in *expr*:

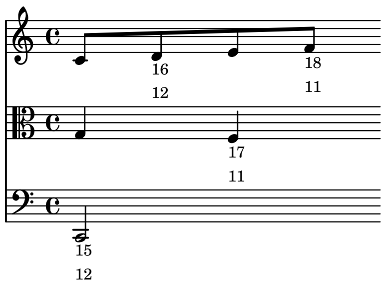
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_diatonic_intervals(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \small
                    \column
                        {
                            16
                            12
                        }
                }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            18
                            11
                        }
                }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \small
                    \column
                        {
                            17
                            11
                        }
                }
    }
```

```
        \new Staff {
            \clef "bass"
            c,2
                _ \markup {
                    \small
                        \column
                            {
                                15
                                12
                            }
                    }
        }
    >>
```

```
>>> show(score)
```



Return none.

11.1.32 labeltools.label_vertical_moments_in_expr_with_interval_class_vectors

labeltools.**label_vertical_moments_in_expr_with_interval_class_vectors**(*expr*, *markup_direction=Down*)

New in version 2.0. Label interval-class vector of every vertical moment in *expr*:

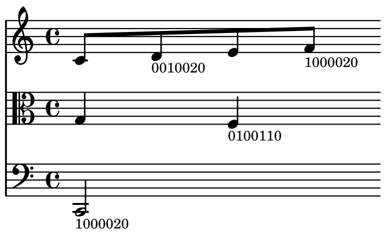
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_interval_class_vectors(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \tiny
                    0010020
                }
        e'8
        f'8
            _ \markup {
                \tiny
                    1000020
                }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \tiny
                    0100110
```

```
                }
        }
        \new Staff {
            \clef "bass"
            c,2
                _ \markup {
                    \tiny
                        1000020
                    }
        }
    >>
```

```
>>> show(score)
```



Return none.

11.1.33 labeltools.label_vertical_moments_in_expr_with_numbered_chromatic_pitch_classe

labeltools.**label_vertical_moments_in_expr_with_numbered_chromatic_pitch_classes**(*expr*, *markup_di*

New in version 2.0. Label pitch-classes of every vertical moment in *expr*:

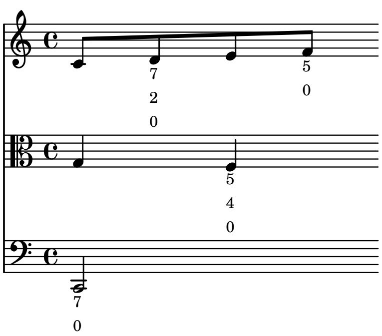
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_numbered_chromatic_pitch_classes(
...     score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \small
                    \column
                        {
                            7
                            2
                            0
                        }
                }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            5
                            0
                        }
                }
    }
    \new Staff {
```

```
            \clef "alto"
            g4
            f4
                _ \markup {
                    \small
                        \column
                            {
                                5
                                4
                                0
                            }
                    }
        }
        \new Staff {
            \clef "bass"
            c,2
                _ \markup {
                    \small
                        \column
                            {
                                7
                                0
                            }
                    }
        }
    >>
```

```
>>> show(score)
```



Return none.

11.1.34 labeltools.label_vertical_moments_in_expr_with_pitch_numbers

`labeltools.`**`label_vertical_moments_in_expr_with_pitch_numbers`**(*expr*, *markup_direction=Down*)

New in version 2.0. Label pitch numbers of every vertical moment in *expr*:

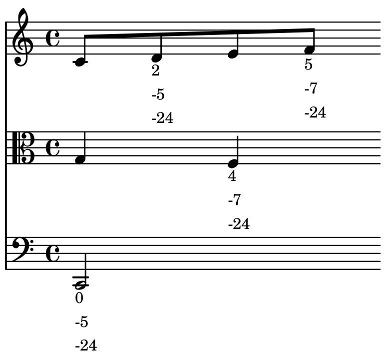
```
>>> score = Score([])
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "alto" g4 f4""")
>>> score.append(staff)
>>> staff = Staff(r"""\clef "bass" c,2""")
>>> score.append(staff)
```

```
>>> labeltools.label_vertical_moments_in_expr_with_pitch_numbers(score)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
            _ \markup {
                \small
                    \column
```

```
                        {
                            2
                            -5
                            -24
                        }
                    }
        e'8
        f'8
            _ \markup {
                \small
                    \column
                        {
                            5
                            -7
                            -24
                        }
                    }
    }
    \new Staff {
        \clef "alto"
        g4
        f4
            _ \markup {
                \small
                    \column
                        {
                            4
                            -7
                            -24
                        }
                    }
    }
    \new Staff {
        \clef "bass"
        c,2
            _ \markup {
                \small
                    \column
                        {
                            0
                            -5
                            -24
                        }
                    }
    }
>>
```

```
>>> show(score)
```



Return none.

11.1.35 labeltools.remove_markup_from_leaves_in_expr

`labeltools.`**`remove_markup_from_leaves_in_expr`**(*expr*)

New in version 1.1. Remove markup from leaves in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> labeltools.label_leaves_in_expr_with_pitch_class_numbers(staff)
>>> f(staff)
\new Staff {
    c'8 _ \markup { \small 0 }
    d'8 _ \markup { \small 2 }
    e'8 _ \markup { \small 4 }
    f'8 _ \markup { \small 5 }
}
```

```
>>> show(staff)
```


```
>>> labeltools.remove_markup_from_leaves_in_expr(staff)
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

Return none.

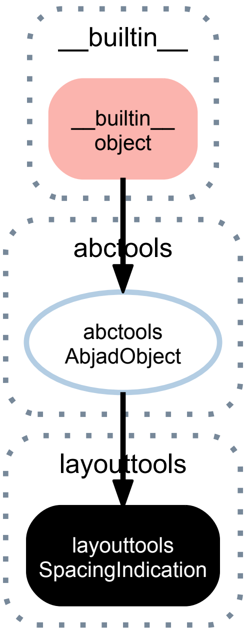
CHAPTER

TWELVE

LAYOUTTOOLS

12.1 Concrete Classes

12.1.1 layouttools.SpacingIndication



class `layouttools.`**`SpacingIndication`**`(*args)`

Spacing indication token.

LilyPond `Score.proportionalNotationDuration` will equal `proportional_notation_duration` when tempo equals `tempo_indication`.

Initialize from tempo mark and proportional notation duration:

```
>>> tempo = contexttools.TempoMark(Duration(1, 8), 44)
>>> indication = layouttools.SpacingIndication(tempo, Duration(1, 68))
```

```
>>> indication
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from constants:

```
>>> layouttools.SpacingIndication(((1, 8), 44), (1, 68))
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Initialize from other spacing indication:

```
>>> layouttools.SpacingIndication(indication)
SpacingIndication(TempoMark(Duration(1, 8), 44), Duration(1, 68))
```

Spacing indications are immutable.

Read-only properties

SpacingIndication.**normalized_spacing_duration**
Read-only proportional notation duration at 60 MM.

SpacingIndication.**proportional_notation_duration**
LilyPond proportional notation duration context setting.

SpacingIndication.**storage_format**
Storage format of Abjad object.

Return string.

SpacingIndication.**tempo_indication**
Abjad tempo indication object.

Special methods

SpacingIndication.**__eq__**(*expr*)
Spacing indications compare equal when normalized spacing durations compare equal.

SpacingIndication.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SpacingIndication.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SpacingIndication.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SpacingIndication.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SpacingIndication.**__ne__**(*expr*)
Spacing indications compare unequal when normalized spacing durations compare unequal.

SpacingIndication.**__repr__**()
Interpreter representation of Abjad object.

Return string.

12.2 Functions

12.2.1 layouttools.make_spacing_vector

layouttools.**make_spacing_vector**(*basic_distance*, *minimum_distance*, *padding*, *stretchability*)
New in version 2.0. Make spacing vector:

```
>>> vector = layouttools.make_spacing_vector(0, 0, 12, 0)
```

Use to set paper block spacing attributes:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
>>> spacing_vector = layouttools.make_spacing_vector(0, 0, 12, 0)
>>> lilypond_file.paper_block.system_system_spacing = spacing_vector
```

```
>>> f(lilypond_file)
% Abjad revision 4229
% 2011-04-07 15:19

\version "2.13.44"
\include "english.ly"
\include "/abjad/trunk/abjad/cfg/abjad.scm"

\paper {
    system-system-spacing = #'(
        (basic-distance . 0) (minimum-distance . 0) (padding . 12) (stretchability . 0))
}

\score {
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
}
```

Return scheme vector.

12.2.2 layouttools.set_line_breaks_cyclically_by_line_duration_ge

layouttools.**set_line_breaks_cyclically_by_line_duration_ge**(*expr*, *line_duration*, *klass=None*, *adjust_eol=False*, *add_empty_bars=False*)

Iterate *klass* instances in *expr* and accumulate prolated duration. Add line break after every total less than or equal to *line_duration*:

```
>>> t = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 4)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
```

```
>>> f(t)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
    }
}
```

```
>>> layouttools.set_line_breaks_cyclically_by_line_duration_ge(t, Duration(4, 8))
>>> f(t)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
        \break
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
        \break
    }
}
```

When `klass=None` set *klass* to measure.

Set *adjust_eol* to `True` to include a magic Scheme incantation to move end-of-line LilyPond TimeSignature and BarLine grobs to the right.

12.2.3 layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge

`layouttools.`**`set_line_breaks_cyclically_by_line_duration_in_seconds_ge`**(*expr*, *line_duration*, *klass=None*, *adjust_eol=False*, *add_empty_bars=False*)

Iterate *klass* instances in *expr* and accumulate duration in seconds. Add line break after every total less than or equal to *line_duration*:

```
>>> t = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 4)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
>>> tempo_mark = contexttools.TempoMark(Duration(1, 8), 44, target_context = Staff)(t)
```

```
>>> f(t)
\new Staff {
    \tempo 8=44
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
    }
}
```

```
>>> layouttools.set_line_breaks_cyclically_by_line_duration_in_seconds_ge(t, Duration(6))
>>> f(t)
```

```
\new Staff {
    \tempo 8=44
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
        \break
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
    }
}
```

When `klass=None` set *klass* to measure.

Set `adjust_eol = True` to include a magic Scheme incantation to move end-of-line LilyPond TimeSignature and BarLine grobs to the right.

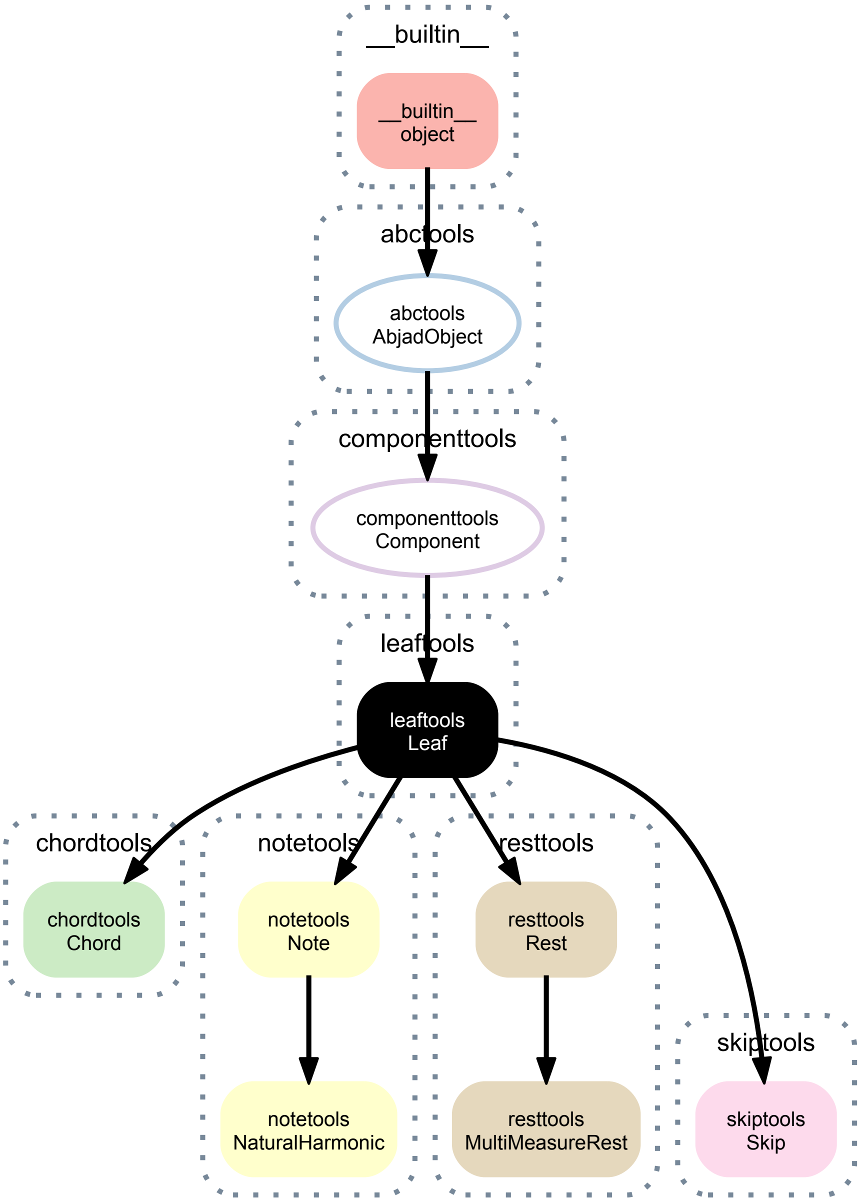
CHAPTER

THIRTEEN

LEAFTOOLS

13.1 Concrete Classes

13.1.1 leaftools.Leaf



class `leaftools.`**`Leaf`**(*written_duration*, *duration_multiplier=None*)

Read-only properties

`Leaf.`**descendants**
Read-only reference to component descendants score selection.

`Leaf.`**duration**

`Leaf.`**duration_in_seconds**

`Leaf.`**leaf_index**

`Leaf.`**lilypond_format**

`Leaf.`**lineage**
Read-only reference to component lineage score selection.

`Leaf.`**multiplied_duration**

`Leaf.`**override**
Read-only reference to LilyPond grob override component plug-in.

`Leaf.`**parent**

`Leaf.`**parentage**
Read-only reference to component parentage score selection.

`Leaf.`**preprolated_duration**

`Leaf.`**prolation**

`Leaf.`**set**
Read-only reference LilyPond context setting component plug-in.

`Leaf.`**spanners**
Read-only reference to unordered set of spanners attached to component.

`Leaf.`**storage_format**
Storage format of Abjad object.

Return string.

`Leaf.`**timespan**
Read-only timespan of component.

`Leaf.`**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

`Leaf.`**duration_multiplier**

`Leaf.`**written_duration**

`Leaf.`**written_pitch_indication_is_at_sounding_pitch**

`Leaf.`**written_pitch_indication_is_nonsemantic**

Special methods

`Leaf.`**__and__**(*arg*)

`Leaf.`**__copy__**(**args*)

`Leaf.`**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`Leaf.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Leaf.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Leaf.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Leaf.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Leaf.__mul__(n)`

`Leaf.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`Leaf.__or__(arg)`

`Leaf.__repr__()`

`Leaf.__rmul__(n)`

`Leaf.__str__()`

`Leaf.__sub__(arg)`

`Leaf.__xor__(arg)`

13.2 Functions

13.2.1 leaftools.all_are_leaves

`leaftools.all_are_leaves(expr)`
New in version 2.6. True when *expr* is a sequence of Abjad leaves:

```
>>> leaves = [Note("c'4"), Rest('r4'), Note("d'4")]
```

```
>>> leaftools.all_are_leaves(leaves)
True
```

True when *expr* is an empty sequence:

```
>>> leaftools.all_are_leaves([])
True
```

Otherwise false:

```
>>> leaftools.all_are_leaves('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

13.2.2 leaftools.change_written_leaf_duration_and_preserve_preprolated_leaf_duration

leaftools.**change_written_leaf_duration_and_preserve_preprolated_leaf_duration**(*leaf*, *written_duration*)

New in version 1.1. Change *leaf* written duration to *written_duration* and preserve preprolated *leaf* duration:

```
>>> note = Note("c'4")
```

```
>>> note.written_duration
Duration(1, 4)
>>> note.preprolated_duration
Duration(1, 4)
```

```
>>> leaftools.change_written_leaf_duration_and_preserve_preprolated_leaf_duration(
...     note, Duration(3, 16))
Note("c'8. * 4/3")
```

```
>>> note.written_duration
Duration(3, 16)
>>> note.preprolated_duration
Duration(1, 4)
```

Add LilyPond multiplier where necessary.

Return *leaf*.

13.2.3 leaftools.copy_written_duration_and_multiplier_from_leaf_to_leaf

leaftools.**copy_written_duration_and_multiplier_from_leaf_to_leaf**(*source_leaf*, *target_leaf*)

New in version 2.0. Copy written duration and multiplier from *source_leaf* to *target_leaf*:

```
>>> note = Note("c'4")
>>> note.duration_multiplier = Multiplier(1, 2)
>>> rest = Rest((1, 64))
>>> leaftools.copy_written_duration_and_multiplier_from_leaf_to_leaf(note, rest)
Rest('r4 * 1/2')
```

Return *target_leaf*.

13.2.4 leaftools.divide_leaf_meiotically

leaftools.**divide_leaf_meiotically**(*leaf*, *n=2*)

New in version 1.1. Divide *leaf* meiotically *n* times:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> leaftools.divide_leaf_meiotically(staff[0], n=4)
```

```
>>> f(staff)
\new Staff {
    c'32 [
    c'32
```

```
    c'32
    c'32
    d'8
    e'8
    f'8 ]
}
```

Replace *leaf* with *n* new leaves.

Preserve parentage and spanners.

Allow divisions into only `1, 2, 4, 8, 16, ...` and other nonnegative integer powers of `2`.

Produce only leaves and never tuplets or other containers.

Return none.

13.2.5 leaftools.divide_leaves_in_expr_meiotically

`leaftools.`**`divide_leaves_in_expr_meiotically`**`(`*expr*, *n=2*`)`

New in version 1.1. Divide leaves meiotically in *expr* *n* times:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> leaftools.divide_leaves_in_expr_meiotically(staff[2:], n=4)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'32
    e'32
    e'32
    e'32
    f'32
    f'32
    f'32
    f'32 ]
}
```

Replace every leaf in *expr* with *n* new leaves.

Preserve parentage and spanners.

Allow divisions into only `1, 2, 4, 8, 16, ...` and other nonnegative integer powers of `2`.

Produce only leaves and never tuplets or other containers.

Return none.

13.2.6 leaftools.expr_has_leaf_with_dotted_written_duration

`leaftools.`**`expr_has_leaf_with_dotted_written_duration`**`(`*expr*`)`

New in version 2.0. True when *expr* has at least one leaf with dotted writtern duration:

```
>>> notes = notetools.make_notes([0], [(1, 16), (2, 16), (3, 16)])
>>> leaftools.expr_has_leaf_with_dotted_written_duration(notes)
True
```

False otherwise:

```
>>> notes = notetools.make_notes([0], [(1, 16), (2, 16), (4, 16)])
>>> leaftools.expr_has_leaf_with_dotted_written_duration(notes)
False
```

Return boolean.

13.2.7 leaftools.fuse_leaves

leaftools.**fuse_leaves**(*leaves*)
New in version 1.1. Fuse thread-contiguous *leaves*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> leaftools.fuse_leaves(staff[1:])
[Note("d'4.")]
>>> f(staff)
\new Staff {
    c'8
    d'4.
}
```

Rewrite duration of first leaf in *leaves*.

Detach all leaves in *leaves* other than first leaf from score.

Return list of first leaf in *leaves*.

13.2.8 leaftools.fuse_leaves_in_container_once_by_counts

leaftools.**fuse_leaves_in_container_once_by_counts**(*container*, *counts*, *klass=None*, *decrease_durations_monotonically=True*)
Fuse leaves in *container* once by *counts* into instances of *klass*.

13.2.9 leaftools.fuse_leaves_in_tie_chain_by_immediate_parent

leaftools.**fuse_leaves_in_tie_chain_by_immediate_parent**(*tie_chain*)
New in version 1.1. Fuse leaves in *tie_chain* by immediate parent:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> tietools.TieSpanner(staff.leaves)
TieSpanner(c'8, c'8, c'8, c'8)
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 ~
        c'8 ~
    }
    {
        c'8 ~
        c'8
    }
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff.leaves[0])
>>> leaftools.fuse_leaves_in_tie_chain_by_immediate_parent(tie_chain)
[[Note("c'4")], [Note("c'4")]]
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'4 ~
```

```
        }
        {
            c'4
        }
    }
```

Return list of fused notes by parent.

13.2.10 leaftools.fuse_tied_leaves_in_components_once_by_durations_without_overhang

leaftools.**fuse_tied_leaves_in_components_once_by_durations_without_overhang**(*components*, *durations*)

New in version 1.1. Fuse tied leaves in *components* once by prolated *durations* without overhang:

```
>>> staff = Staff(notetools.make_repeated_notes(8))
>>> tietools.TieSpanner(staff.leaves)
TieSpanner(c'8, c'8, c'8, c'8, c'8, c'8, c'8, c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8 ~
    c'8 ~
    c'8 ~
    c'8 ~
    c'8 ~
    c'8 ~
    c'8
}
```

```
>>> leaftools.fuse_tied_leaves_in_components_once_by_durations_without_overhang(
... staff, [Duration(3, 8), Duration(3, 8)])
```

```
>>> f(staff)
\new Staff {
    c'4. ~
    c'4. ~
    c'8 ~
    c'8
}
```

Return none.

13.2.11 leaftools.get_composite_offset_difference_series_from_leaves_in_expr

leaftools.**get_composite_offset_difference_series_from_leaves_in_expr**(*expr*)

New in version 2.0. Get composite offset difference series from leaves in *expr*:

```
>>> staff_1 = Staff(r"\times 4/3 { c'8 d'8 e'8 }")
>>> staff_2 = Staff("f'8 g'8 a'8 b'8")
>>> score = Score([staff_1, staff_2])
```

```
>>> f(score)
    \new Score <<
        \new Staff {
            \fraction \times 4/3 {
                c'8
                d'8
                e'8
            }
        }
        \new Staff {
            f'8
            g'8
```

```
            a'8
            b'8
        }
    >>
```

```
>>> for x in leaftools.get_composite_offset_difference_series_from_leaves_in_expr(score):
...     x
...
Duration(1, 8)
Duration(1, 24)
Duration(1, 12)
Duration(1, 12)
Duration(1, 24)
Duration(1, 8)
```

Composite offset difference series defined equal to time intervals between unique start and stop offsets of leaves in *expr*.

Return list of durations.

13.2.12 leaftools.get_composite_offset_series_from_leaves_in_expr

leaftools.**get_composite_offset_series_from_leaves_in_expr**(*expr*)
New in version 2.0. Get composite offset series from leaves in *expr*:

```
>>> staff_1 = Staff(r"\times 4/3 { c'8 d'8 e'8 }")
>>> staff_2 = Staff("f'8 g'8 a'8 b'8")
>>> score = Score([staff_1, staff_2])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \fraction \times 4/3 {
            c'8
            d'8
            e'8
        }
    }
    \new Staff {
        f'8
        g'8
        a'8
        b'8
    }
>>
```

```
>>> for offset in leaftools.get_composite_offset_series_from_leaves_in_expr(score):
...     offset
...
Offset(0, 1)
Offset(1, 8)
Offset(1, 6)
Offset(1, 4)
Offset(1, 3)
Offset(3, 8)
Offset(1, 2)
```

Equal to list of unique start and stop offsets of leaves in *expr*.

Return list of fractions.

13.2.13 leaftools.get_leaf_at_index_in_measure_number_in_expr

leaftools.**get_leaf_at_index_in_measure_number_in_expr**(*expr*, *measure_number*, *leaf_index*)
New in version 2.0. Get leaf at *leaf_index* in *measure_number* in *expr*:

```
>>> t = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
>>> f(t)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
```

```
>>> leaftools.get_leaf_at_index_in_measure_number_in_expr(t, 2, 0)
Note("e'8")
```

Return leaf or none.

13.2.14 leaftools.get_leaf_in_expr_with_maximum_duration

leaftools.**get_leaf_in_expr_with_maximum_duration**(*expr*)

New in version 2.5. Get leaf in *expr* with maximum prolated duration:

```
>>> staff = Staff("c'4.. d'16 e'4.. f'16")
```

```
>>> leaftools.get_leaf_in_expr_with_maximum_duration(staff)
Note("c'4..")
```

When two leaves in *expr* are both of equally maximal prolated duration, return the first leaf encountered in forward iteration.

Return none when *expr* contains no leaves:

```
>>> leaftools.get_leaf_in_expr_with_maximum_duration([]) is None
True
```

Return leaf.

13.2.15 leaftools.get_leaf_in_expr_with_minimum_duration

leaftools.**get_leaf_in_expr_with_minimum_duration**(*expr*)

New in version 2.5. Get leaf in *expr* with minimum prolated duration:

```
>>> staff = Staff("c'4.. d'16 e'4.. f'16")
```

```
>>> leaftools.get_leaf_in_expr_with_minimum_duration(staff)
Note("d'16")
```

When two leaves in *expr* are both of equally minimal prolated duration, return the first leaf encountered in forward iteration.

Return none when *expr* contains no leaves:

```
>>> leaftools.get_leaf_in_expr_with_minimum_duration([]) is None
True
```

Return leaf.

13.2.16 leaftools.get_nth_leaf_in_expr

`leaftools.get_nth_leaf_in_expr(expr, n=0)`

New in version 2.0. Get *n* th leaf in *expr*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
```

```
>>> for n in range(6):
...     leaftools.get_nth_leaf_in_expr(staff, n)
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("f'8")
Note("g'8")
Note("a'8")
```

Read backwards for negative values of *n*.

```
>>> leaftools.get_nth_leaf_in_expr(staff, -1)
Note("a'8")
```

Note: Because this function returns as soon as it finds instance *n* of *klasses*, it is more efficient to call `leaftools.get_nth_leaf_in_expr(expr, 0)` than `expr.leaves[0]`. It is likewise more efficient to call `leaftools.get_nth_leaf_in_expr(expr, -1)` than `expr.leaves[-1]`.

Return leaf of none.

13.2.17 leaftools.get_nth_leaf_in_thread_from_leaf

`leaftools.get_nth_leaf_in_thread_from_leaf(leaf, n=0)`

New in version 2.0. Get *n* th leaf in thread from *leaf*:

```
>>> staff = Staff(2 * Voice("c'8 d'8 e'8 f'8"))
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    \new Voice {
        c'8
        d'8
        e'8
        f'8
    }
    \new Voice {
        g'8
        a'8
        b'8
        c''8
```

```
        }
    }
```

```
>>> for n in range(8):
...     print n, leaftools.get_nth_leaf_in_thread_from_leaf(staff[0][0], n)
...
0 c'8
1 d'8
2 e'8
3 f'8
4 None
5 None
6 None
7 None
```

Return leaf or none.

13.2.18 leaftools.is_bar_line_crossing_leaf

leaftools.**is_bar_line_crossing_leaf**(*leaf*)

New in version 2.0. True when *leaf* crosses bar line:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> t[2].written_duration *= 2
>>> contexttools.TimeSignatureMark((2, 8), partial = Duration(1, 8))(t[2])
TimeSignatureMark((2, 8), partial=Duration(1, 8))(e'4)
>>> f(t)
\new Staff {
    c'8
    d'8
    \partial 8
    \time 2/8
    e'4
    f'8
}
>>> leaftools.is_bar_line_crossing_leaf(t.leaves[2])
True
```

Otherwise false:

```
>>> leaftools.is_bar_line_crossing_leaf(t.leaves[3])
False
```

Return boolean.

13.2.19 leaftools.list_durations_of_leaves_in_expr

leaftools.**list_durations_of_leaves_in_expr**(*expr*)

New in version 2.0. List prolated durations of leaves in *expr*:

```
>>> staff = Staff(r"    imes 2/3 { c'8 d'8 e'8 }        imes 2/3 { c'8 d'8 e'8 }")
```

```
>>> for duration in leaftools.list_durations_of_leaves_in_expr(staff):
...     duration
...
Duration(1, 12)
Duration(1, 12)
Duration(1, 12)
Duration(1, 12)
Duration(1, 12)
Duration(1, 12)
```

Return list of durations.

13.2.20 leaftools.list_written_durations_of_leaves_in_expr

leaftools.**list_written_durations_of_leaves_in_expr**(*expr*)
New in version 2.0. List the written durations of leaves in *expr*:

```
>>> staff = Staff(tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8") * 2)
```

```
>>> for duration in leaftools.list_written_durations_of_leaves_in_expr(staff):
...     duration
...
Duration(1, 8)
Duration(1, 8)
Duration(1, 8)
Duration(1, 8)
Duration(1, 8)
 Duration(1, 8)
```

Return list of durations.

13.2.21 leaftools.make_leaves

leaftools.**make_leaves**(*pitches*, *durations*, *decrease_durations_monotonically=True*, *tie_rests=False*, *forbidden_written_duration=None*, *metrical_hiearchy=None*)
New in version 1.1. Make leaves.

Example 1. Integer and string elements in *pitches* result in notes:

```
>>> leaves = leaftools.make_leaves([2, 4, 'F#5', 'G#5'], [Duration(1, 4)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    d'4
    e'4
    fs''4
    gs''4
}
```

```
>>> show(staff)
```

Example 2. Tuple elements in *pitches* result in chords:

```
>>> leaves = leaftools.make_leaves([(0, 2, 4), ('F#5', 'G#5', 'A#5')], [Duration(1, 2)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    <c' d' e'>2
    <fs'' gs'' as''>2
}
```

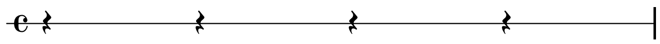
```
>>> show(staff)
```

Example 3. None-valued elements in *pitches* result in rests:

```
>>> leaves = leaftools.make_leaves([None, None, None, None], [Duration(1, 4)])
>>> staff = stafftools.RhythmicStaff(leaves)
```

```
>>> f(staff)
\new RhythmicStaff {
    r4
    r4
    r4
    r4
}
```

```
>>> show(staff)
```

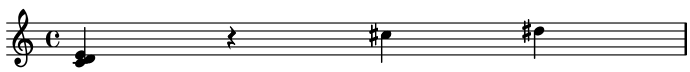


Example 4. You can mix and match values passed to *pitches*:

```
>>> leaves = leaftools.make_leaves([(0, 2, 4), None, ’C#5’, ’D#5’], [Duration(1, 4)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    <c’ d’ e’>4
    r4
    cs’’4
    ds’’4
}
```

```
>>> show(staff)
```

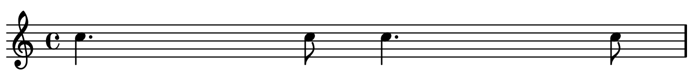


Example 5. Read *pitches* cyclically when the length of *pitches* is less than the length of *durations*:

```
>>> leaves = leaftools.make_leaves([’C5’], 2 * [Duration(3, 8), Duration(1, 8)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    c’’4.
    c’’8
    c’’4.
    c’’8
}
```

```
>>> show(staff)
```

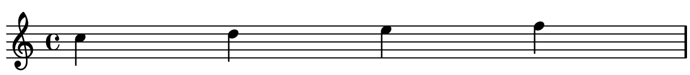


Example 6. Read *durations* cyclically when the length of *durations* is less than the length of *pitches*:

```
>>> leaves = leaftools.make_leaves("c’’ d’’ e’’ f’’", [Duration(1, 4)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    c’’4
    d’’4
    e’’4
    f’’4
}
```

```
>>> show(staff)
```

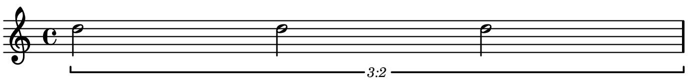


Example 7. Elements in *durations* with non-power-of-two denominators result in tuplet-nested leaves:

```
>>> leaves = leaftools.make_leaves(['D5'], 3 * [Duration(1, 3)])
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        d''2
        d''2
        d''2
    }
}
```

```
>>> show(staff)
```

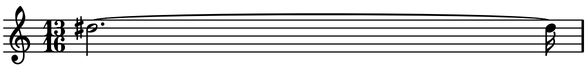


Example 8. Set *decrease_durations_monotonically* to true to return nonassignable durations tied from greatest to least:

```
>>> leaves = leaftools.make_leaves(['D#5'], [Duration(13, 16)])
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((13, 16))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 13/16
    ds''2. ~
    ds''16
}
```

```
>>> show(staff)
```



Example 9. Set *decrease_durations_monotonically* to false to return nonassignable durations tied from least to greatest:

```
>>> leaves = leaftools.make_leaves(['E5'], [Duration(13, 16)],
...     decrease_durations_monotonically=False)
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((13, 16))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 13/16
    e''16 ~
    e''2.
}
```

```
>>> show(staff)
```

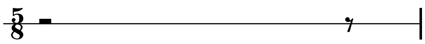
Example 10. Set *tie_rests* to true to return tied rests for nonassignable durations. Note that LilyPond does not engrave ties between rests:

```
>>> leaves = leaftools.make_leaves([None], [Duration(5, 8)], tie_rests=True)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 8))(staff)
```

```
>>> f(staff)
\new RhythmicStaff {
    \time 5/8
    r2 ~
```

```
    r8
}
```

```
>>> show(staff)
```

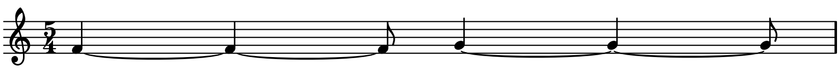


Example 11. Set *forbidden_written_duration* to avoid notes greater than or equal to a certain written duration:

```
>>> leaves = leaftools.make_leaves("f' g'", [Duration(5, 8)],
...     forbidden_written_duration=Duration(1, 2))
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 5/4
    f'4 ~
    f'4 ~
    f'8
    g'4 ~
    g'4 ~
    g'8
}
```

```
>>> show(staff)
```

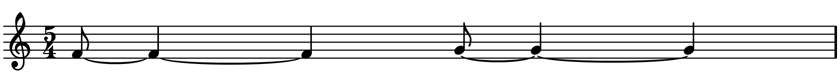


Example 12. You may set *forbidden_written_duration* and *decrease_durations_monotonically* together:

```
>>> leaves = leaftools.make_leaves("f' g'", [Duration(5, 8)],
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=False)
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 5/4
    f'8 ~
    f'4 ~
    f'4
    g'8 ~
    g'4 ~
    g'4
}
```

```
>>> show(staff)
```



Return list of leaves.

13.2.22 leaftools.make_leaves_from_talea

`leaftools.`**`make_leaves_from_talea`**(*talea*, *talea_denominator*, *decrease_durations_monotonically=True*, *tie_rests=False*, *forbidden_written_duration=None*)

New in version 2.0. Make leaves from *talea*.

Interpret positive elements in *talea* as notes numerators.

Interpret negative elements in *talea* as rests numerators.

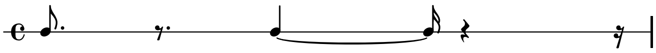
Set the pitch of all notes to middle C.

Example 1. Make leaves from talea:

```
>>> leaves = leaftools.make_leaves_from_talea([3, -3, 5, -5], 16)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
```

```
>>> f(staff)
\new RhythmicStaff {
    \time 4/4
    c'8.
    r8.
    c'4 ~
    c'16
    r4
    r16
}
```

```
>>> show(staff)
```

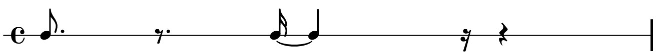


Example 2. Increase durations monotonically:

```
>>> leaves = leaftools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     decrease_durations_monotonically=False)
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
```

```
>>> f(staff)
\new RhythmicStaff {
    \time 4/4
    c'8.
    r8.
    c'16 ~
    c'4
    r16
    r4
}
```

```
>>> show(staff)
```



Example 3. Forbid written durations greater than or equal to a half note:

```
>>> leaves = leaftools.make_leaves_from_talea(
...     [3, -3, 5, -5], 16,
...     forbidden_written_duration=Duration(1, 4))
>>> staff = stafftools.RhythmicStaff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))(staff)
```

```
>>> f(staff)
\new RhythmicStaff {
    \time 4/4
    c'8.
    r8.
    c'8 ~
    c'8 ~
    c'16
    r8
    r8
    r16
}
```

```
>>> show(staff)
```

Return list of leaves.

13.2.23 leaftools.make_tied_leaf

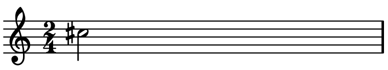
`leaftools.make_tied_leaf`(*kind*, *duration*, *decrease_durations_monotonically=True*, *forbidden_written_duration=None*, *pitches=None*, *tie_parts=True*)

New in version 1.0. Example 1. Make note:

```
>>> leaves = leaftools.make_tied_leaf(notetools.Note, Duration(1, 2), pitches='C#5')
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((2, 4))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    cs''2
}
```

```
>>> show(staff)
```

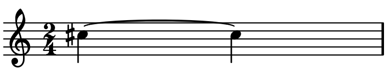


Example 2. Make note and forbid half notes:

```
>>> leaves = leaftools.make_tied_leaf(
...     notetools.Note, Duration(1, 2), pitches='C#5',
...     forbidden_written_duration=Duration(1, 2))
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((2, 4))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    cs''4 ~
    cs''4
}
```

```
>>> show(staff)
```



Example 3. Make tied note with half notes forbidden and durations decreasing monotonically:

```
>>> leaves = leaftools.make_tied_leaf(
...     notetools.Note, Duration(9, 8), pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=True)
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((9, 8))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 9/8
    cs''4 ~
    cs''4 ~
    cs''4 ~
    cs''4 ~
    cs''8
}
```

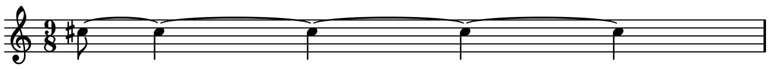
```
>>> show(staff)
```

Example 4. Make tied note with half notes forbidden and durations increasing monotonically:

```
>>> leaves = leaftools.make_tied_leaf(
...     notetools.Note, Duration(9, 8), pitches='C#5',
...     forbidden_written_duration=Duration(1, 2),
...     decrease_durations_monotonically=False)
>>> staff = Staff(leaves)
>>> time_signature = contexttools.TimeSignatureMark((9, 8))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 9/8
    cs''8 ~
    cs''4 ~
    cs''4 ~
    cs''4 ~
    cs''4
}
```

```
>>> show(staff)
```



Return list of leaves.

13.2.24 leaftools.remove_initial_rests_from_sequence

leaftools.**remove_initial_rests_from_sequence**(*sequence*)
New in version 2.0. Remove initial rests from *sequence*:

```
>>> staff = Staff("r8 r8 c'8 d'8 r4 r4")
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
    r4
}
```

```
>>> leaftools.remove_initial_rests_from_sequence(staff)
[Note("c'8"), Note("d'8"), Rest('r4'), Rest('r4')]
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
    r4
}
```

Return list.

13.2.25 leaftools.remove_leaf_and_shrink_durated_parent_containers

leaftools.**remove_leaf_and_shrink_durated_parent_containers**(*leaf*)
New in version 1.1. Remove *leaf* and shrink durated parent containers:

```
>>> measure = Measure((4, 8), [])
>>> measure.append(tuplettools.FixedDurationTuplet((2, 8), "c'8 d'8 e'8"))
>>> measure.append(tuplettools.FixedDurationTuplet((2, 8), "f'8 g'8 a'8"))
>>> beamtools.BeamSpanner(measure.leaves)
BeamSpanner(c'8, d'8, e'8, f'8, g'8, a'8)
```

```
>>> f(measure)
{
    \time 4/8
    \times 2/3 {
        c'8 [
        d'8
        e'8
    }
    \times 2/3 {
        f'8
        g'8
        a'8 ]
    }
}
```

```
>>> leaftools.remove_leaf_and_shrink_durated_parent_containers(measure.leaves[0])
```

```
>>> f(measure)
{
    \time 5/12
    \scaleDurations #'(2 . 3) {
        {
            d'8 [
            e'8
        }
        {
            f'8
            g'8
            a'8 ]
        }
    }
}
```

Return none.

13.2.26 leaftools.remove_outer_rests_from_sequence

leaftools.**remove_outer_rests_from_sequence**(*sequence*)

New in version 2.0. Remove outer rests from *sequence*:

```
>>> staff = Staff("r8 r8 c'8 d'8 r4 r4")
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
    r4
}
```

```
>>> leaftools.remove_outer_rests_from_sequence(staff)
[Note("c'8"), Note("d'8")]
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
```

```
        r4
    }
```

Return list.

13.2.27 leaftools.remove_terminal_rests_from_sequence

leaftools.**remove_terminal_rests_from_sequence**(*sequence*)

New in version 2.0. Remove terminal rests from *sequence*:

```
>>> staff = Staff("r8 r8 c'8 d'8 r4 r4")
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
    r4
}
```

```
>>> leaftools.remove_terminal_rests_from_sequence(staff)
[Rest('r8'), Rest('r8'), Note("c'8"), Note("d'8")]
```

```
>>> f(staff)
\new Staff {
    r8
    r8
    c'8
    d'8
    r4
    r4
}
```

Return list.

13.2.28 leaftools.repeat_leaf

leaftools.**repeat_leaf**(*leaf*, *total=1*)

New in version 1.1. Repeat *leaf* and extend spanners:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> leaftools.repeat_leaf(staff[0], total=3)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    c'8
    c'8
    d'8
    e'8
    f'8 ]
}
```

Preserve *leaf* written duration.

Preserve parentage and spanners.

Return none.

13.2.29 leaftools.repeat_leaves_in_expr

leaftools.**repeat_leaves_in_expr**(*expr*, *total=1*)
New in version 1.1. Repeat leaves in *expr* and extend spanners:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> result = leaftools.repeat_leaves_in_expr(staff[2:], total=3)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    e'8
    e'8
    f'8
    f'8
    f'8 ]
}
```

Preserve leaf written durations.

Preserve parentage and spanners.

Return none.

13.2.30 leaftools.replace_leaves_in_expr_with_named_parallel_voices

leaftools.**replace_leaves_in_expr_with_named_parallel_voices**(*expr*, *upper_name*, *lower_name*)
Replace leaves in *expr* with two parallel voices containing copies of leaves in *expr*, with the upper voice named *upper_name* and the lower voice named *lower_name*:

```
>>> c = p('{ c c c c }')
>>> f(c)
{
    c4
    c4
    c4
    c4
}
```

```
>>> leaves = leaftools.replace_leaves_in_expr_with_named_parallel_voices(
... c.leaves[1:3], 'upper', 'lower')
```

```
>>> f(c)
{
    c4
    <<
        \context Voice = "upper" {
```

```
                c4
                c4
            }
            \context Voice = "lower" {
                c4
                c4
            }
        >>
        c4
}
```

If leaves in *expr* have different immediate parents, parallel voices will be created in each parent:

```
>>> c = p(r'{ c8 \times 2/3 { c8 c c } \times 4/5 { c16 c c c c } c8 }')
>>> f(c)
{
    c8
    \times 2/3 {
        c8
        c8
        c8
    }
    \times 4/5 {
        c16
        c16
        c16
        c16
        c16
    }
    c8
}
```

```
>>> leaves = leaftools.replace_leaves_in_expr_with_named_parallel_voices(
... c.leaves[2:7], 'upper', 'lower')
```

```
>>> f(c)
{
    c8
    \times 2/3 {
        c8
        <<
            \context Voice = "upper" {
                c8
                c8
            }
            \context Voice = "lower" {
                c8
                c8
            }
        >>
    }
    \times 4/5 {
        <<
            \context Voice = "upper" {
                c16
                c16
                c16
            }
            \context Voice = "lower" {
                c16
                c16
                c16
            }
        >>
        c16
        c16
    }
    c8
}
```

Returns a list leaves in upper voice, and a list of leaves in lower voice.

13.2.31 leaftools.replace_leaves_in_expr_with_parallel_voices

leaftools.**replace_leaves_in_expr_with_parallel_voices**(*expr*)

Replace leaves in *expr* with two parallel voices containing copies of leaves in *expr*:

```
>>> c = p('{ c c c c }')
>>> f(c)
{
    c4
    c4
    c4
    c4
}
```

```
>>> leaftools.replace_leaves_in_expr_with_parallel_voices(c.leaves[1:3])
([Note('c4'), Note('c4')], [Note('c4'), Note('c4')])
```

```
>>> f(c)
{
    c4
    <<
        \new Voice {
            c4
            c4
        }
        \new Voice {
            c4
            c4
        }
    >>
    c4
}
```

If leaves in *expr* have different immediate parents, parallel voices will be created in each parent:

```
>>> c = p(r'{ c8 \times 2/3 { c8 c c } \times 4/5 { c16 c c c c } c8 }')
>>> f(c)
{
    c8
    \times 2/3 {
        c8
        c8
        c8
    }
    \times 4/5 {
        c16
        c16
        c16
        c16
        c16
    }
    c8
}
```

```
>>> leaves = leaftools.replace_leaves_in_expr_with_parallel_voices(c.leaves[2:7])
```

```
>>> f(c)
{
    c8
    \times 2/3 {
        c8
        <<
            \new Voice {
                c8
                c8
            }
            \new Voice {
                c8
                c8
            }
        >>
```

```
        }
        \times 4/5 {
            <<
                \new Voice {
                    c16
                    c16
                    c16
                }
                \new Voice {
                    c16
                    c16
                    c16
                }
            >>
            c16
            c16
        }
        c8
    }
```

Returns a list leaves in upper voice, and a list of leaves in lower voice.

13.2.32 leaftools.rest_leaf_at_offset

`leaftools.rest_leaf_at_offset(leaf, offset)`

New in version 1.1. Split *leaf* at *offset* and rest right half:

```
>>> staff = Staff("c'8 ( d'8 e'8 f'8 )")
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> leaftools.rest_leaf_at_offset(staff.leaves[1], (1, 32))
([Note("d'32")], [Note("d'16.")])
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'32
    r16.
    e'8
    f'8 )
}
```

Return list of leaves to left of *offset* together with list of leaves to right of *offset*.

13.2.33 leaftools.scale_preprolated_leaf_duration

`leaftools.scale_preprolated_leaf_duration(leaf, multiplier)`

New in version 1.1. Scale preprolated *leaf* leaf duration by dotted *multiplier*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.scale_preprolated_leaf_duration(staff[1], Duration(3, 2))
[Note("d'8.")]
>>> f(staff)
\new Staff {
    c'8 [
    d'8.
    e'8
```

```
        f'8 ]
    }
```

Scale preprolated *leaf* duration by tied *multiplier*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.scale_preprolated_leaf_duration(staff[1], Duration(5, 4))
[Note("d'8"), Note("d'32")]
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ~
    d'32
    e'8
    f'8 ]
}
```

Scale preprolated *leaf* duration by by *multiplier* without power-of-two denominator:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.scale_preprolated_leaf_duration(staff[1], Duration(2, 3))
[Note("d'8")]
>>> f(staff)
\new Staff {
    c'8 [
    \times 2/3 {
        d'8
    }
    e'8
    f'8 ]
}
```

Scale preprolated *leaf* duration by tied *multiplier* without power-of-two denominator:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.scale_preprolated_leaf_duration(staff[1], Duration(5, 6))
[Note("d'8"), Note("d'32")]
>>> f(staff)
\new Staff {
    c'8 [
    \times 2/3 {
        d'8 ~
        d'32
    }
    e'8
    f'8 ]
}
```

Return *leaf*.

13.2.34 leaftools.set_preprolated_leaf_duration

`leaftools.`**`set_preprolated_leaf_duration`**(*leaf*, *new_preprolated_duration*)

New in version 1.1. Set preprolated *leaf* duration:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.set_preprolated_leaf_duration(staff[1], Duration(3, 16))
[Note("d'8.")]
>>> f(staff)
\new Staff {
    c'8 [
    d'8.
```

```
        e'8
        f'8 ]
    }
```

Set tied preprolated *leaf* duration:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.set_preprolated_leaf_duration(staff[1], Duration(5, 32))
[Note("d'8"), Note("d'32")]
>>> f(staff)
\new Staff {
    c'8 [
    d'8 ~
    d'32
    e'8
    f'8 ]
}
```

Set preprolated *leaf* duration without power-of-two denominator:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.set_preprolated_leaf_duration(staff[1], Duration(1, 12))
[Note("d'8")]
>>> f(staff)
\new Staff {
    c'8 [
    \times 2/3 {
        d'8
    }
    e'8
    f'8 ]
}
```

Set preprolated *leaf* duration without power-of-two denominator:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'8, d'8, e'8, f'8)
>>> leaftools.set_preprolated_leaf_duration(staff[1], Duration(5, 48))
[Note("d'8"), Note("d'32")]
>>> f(staff)
\new Staff {
    c'8 [
    \times 2/3 {
        d'8 ~
        d'32
    }
    e'8
    f'8 ]
}
```

Set preprolated *leaf* duration with LilyPond multiplier:

```
>>> note = Note(0, (1, 8))
>>> note.duration_multiplier = Duration(1, 2)
>>> leaftools.set_preprolated_leaf_duration(note, Duration(5, 48))
[Note("c'8 * 5/6")]
>>> f(note)
c'8 * 5/6
```

Return list of *leaf* and leaves newly tied to *leaf*.

13.2.35 leaftools.show_leaves

leaftools.**show_leaves**(*leaves*, *suppress_pdf=False*)

New in version 2.0. Show *leaves* in temporary piano staff score:

```
>>> leaves = leaftools.make_leaves([None, 1, (-24, -22, 7, 21), None], (1, 4))
>>> score = leaftools.show_leaves(leaves)
\new Score <<
    \new PianoStaff <<
        \context Staff = "treble" {
            \clef "treble"
            r4
            cs'4
            <g' a''>4
            r4
        }
        \context Staff = "bass" {
            \clef "bass"
            r4
            r4
            <c, d,>4
            r4
        }
    >>
>>
```

Useful when working with notes, rests, chords not yet added to score.

Return temporary piano staff score.

13.2.36 leaftools.split_leaf_at_offset

leaftools.**split_leaf_at_offset**(*leaf*, *offset*, *fracture_spanners=False*, *tie_split_notes=True*, *tie_split_rests=False*)

Split *leaf* at *offset*.

Example 1. Split note at assignable offset. Two notes result. Do not tie notes:

```
>>> staff = Staff(r"abj: | 2/8 c'8 ( d'8 || 2/8 e'8 f'8 ) |")
>>> beamtools.apply_beam_spanners_to_measures_in_expr(staff)
[BeamSpanner(|2/8(2)|), BeamSpanner(|2/8(2)|)]
>>> contexttools.DynamicMark('f')(staff.leaves[0])
DynamicMark('f')(c'8)
>>> marktools.Articulation('accent')(staff.leaves[0])
Articulation('accent')(c'8)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 -\accent \f [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> leaftools.split_leaf_at_offset(staff.leaves[0], (1, 32),
...     tie_split_notes=False)
([Note("c'32")], [Note("c'16.")])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'32 -\accent \f [ (
        c'16.
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
```

```
    }
}
```

Example 2. Handle grace and after grace containers correctly.

```
>>> staff = Staff(r"abj: | 2/8 c'8 ( d'8 || 2/8 e'8 f'8 ) |")
>>> beamtools.apply_beam_spanners_to_measures_in_expr(staff)
[BeamSpanner(|2/8(2)|), BeamSpanner(|2/8(2)|)]
>>> gracetools.GraceContainer("cs'16")(staff.leaves[0])
Note("c'8")
>>> gracetools.GraceContainer("ds'16", kind='after')(staff.leaves[0])
Note("c'8")
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        \grace {
            cs'16
        }
        \afterGrace
        c'8 [ (
        {
            ds'16
        }
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

```
>>> leaftools.split_leaf_at_offset(staff.leaves[0], (1, 32),
...     tie_split_notes=False)
([Note("c'32")], [Note("c'16.")])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        \grace {
            cs'16
        }
        c'32 [ (
        \afterGrace
        c'16.
        {
            ds'16
        }
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

Return pair.

13.2.37 leaftools.split_leaf_at_offsets

`leaftools.`**`split_leaf_at_offsets`**(*leaf*, *offsets*, *cyclic=False*, *fracture_spanners=False*, *tie_split_notes=True*, *tie_split_rests=False*)

New in version 2.10. Split *leaf* at *offsets*.

Example 1. Split note once at *offsets* and tie split notes:

```
>>> staff = Staff("c'1 ( d'1 )")
```

```
>>> f(staff)
\new Staff {
    c'1 (
    d'1 )
}
```

```
>>> leaftools.split_leaf_at_offsets(staff[0], [(3, 8)],
...     tie_split_notes=True)
[[Note("c'4.")], [Note("c'2"), Note("c'8")]]
```

```
>>> f(staff)
\new Staff {
    c'4. ( ~
    c'2 ~
    c'8
    d'1 )
}
```

Example 2. Split note cyclically at *offsets* and tie split notes:

```
>>> staff = Staff("c'1 ( d'1 )")
```

```
>>> f(staff)
\new Staff {
    c'1 (
    d'1 )
}
```

```
>>> leaftools.split_leaf_at_offsets(staff[0], [(3, 8)], cyclic=True,
...     tie_split_notes=True)
[[Note("c'4.")], [Note("c'4.")], [Note("c'4")]]
```

```
>>> f(staff)
\new Staff {
    c'4. ( ~
    c'4. ~
    c'4
    d'1 )
}
```

Example 3. Split note once at *offsets* and do no tie split notes:

```
>>> staff = Staff("c'1 ( d'1 )")
```

```
>>> f(staff)
\new Staff {
    c'1 (
    d'1 )
}
```

```
>>> leaftools.split_leaf_at_offsets(staff[0], [(3, 8)],
...     tie_split_notes=False)
[[Note("c'4.")], [Note("c'2"), Note("c'8")]]
```

```
>>> f(staff)
\new Staff {
    c'4. (
    c'2 ~
    c'8
    d'1 )
}
```

Example 4. Split note cyclically at *offsets* and do not tie split notes:

```
>>> staff = Staff("c'1 ( d'1 )")
```

```
>>> f(staff)
\new Staff {
    c'1 (
    d'1 )
}
```

```
>>> leaftools.split_leaf_at_offsets(staff[0], [(3, 8)], cyclic=True,
...     tie_split_notes=False)
[[Note("c'4.")], [Note("c'4.")], [Note("c'4")]]
```

```
>>> f(staff)
\new Staff {
    c'4. (
    c'4.
    c'4
    d'1 )
}
```

Example 5. Split tupletted note once at *offsets* and tie split notes:

```
>>> staff = Staff(r"\times 2/3 { c'2 ( d'2 e'2 ) }")
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'2 (
        d'2
        e'2 )
    }
}
```

```
>>> leaftools.split_leaf_at_offsets(staff.leaves[1], [(1, 6)], cyclic=False,
...     tie_split_notes=True)
[[Note("d'4")], [Note("d'4")]]
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'2 (
        d'4 ~
        d'4
        e'2 )
    }
}
```

Note: Add examples showing mark and context mark handling.

Return list of shards.

13.2.38 leaftools.yield_groups_of_mixed_notes_and_chords_in_sequence

leaftools.**yield_groups_of_mixed_notes_and_chords_in_sequence**(*sequence*)
New in version 2.0. Yield groups of mixed notes and chords in *sequence*:

```
>>> staff = Staff("c'8 d'8 r8 r8 <e' g'>8 <f' a'>8 g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    r8
    r8
    <e' g'>8
    <f' a'>8
    g'8
    a'8
```

```
        r8
        r8
        <b' d''>8
        <c'' e''>8
    }
```

```
>>> for group in leaftools.yield_groups_of_mixed_notes_and_chords_in_sequence(staff):
...     group
...
(Note("c'8"), Note("d'8"))
(Chord("<e' g'>8"), Chord("<f' a'>8"), Note("g'8"), Note("a'8"))
(Chord("<b' d''>8"), Chord("<c'' e''>8"))
```

Return generator.

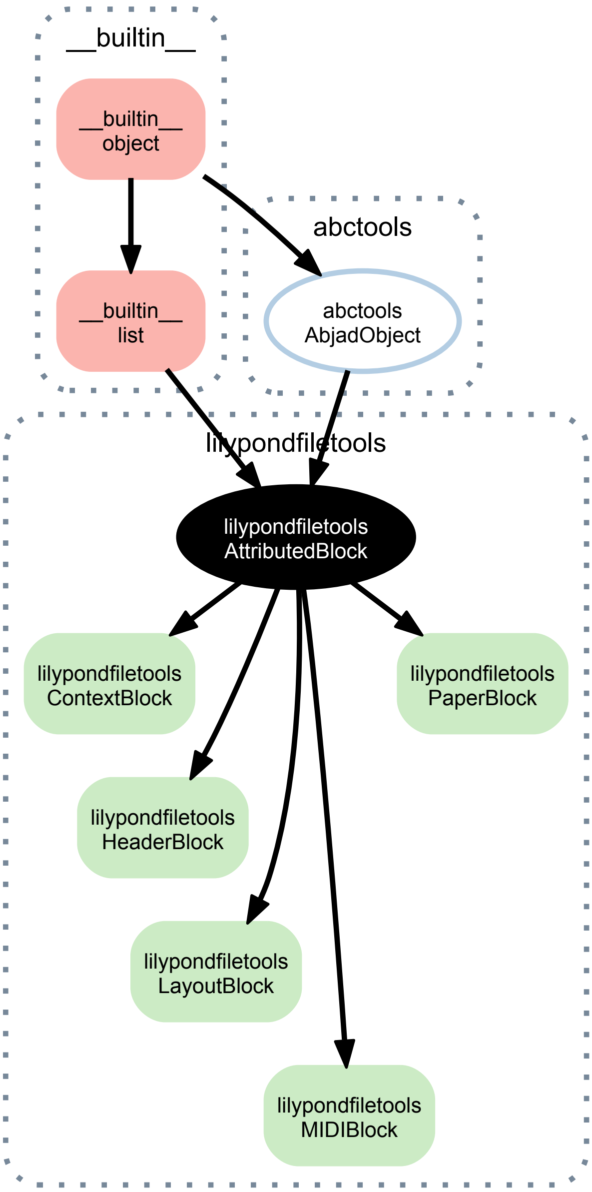
CHAPTER

FOURTEEN

LILYPONDFILETOOLS

14.1 Abstract Classes

14.1.1 lilypondfiletools.AttributedBlock



class lilypondfiletools.**AttributedBlock**
New in version 2.0. Abjad model of LilyPond input file block with attributes.

Read-only properties

AttributedBlock.**lilypond_format**

AttributedBlock.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

AttributedBlock.**is_formatted_when_empty**

Methods

AttributedBlock.**append**()
L.append(object) – append object to end

AttributedBlock.**count**(*value*) → integer – return number of occurrences of value

AttributedBlock.**extend**()
L.extend(iterable) – extend list by appending elements from the iterable

AttributedBlock.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

AttributedBlock.**insert**()
L.insert(index, object) – insert object before index

AttributedBlock.**pop**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

AttributedBlock.**remove**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

AttributedBlock.**reverse**()
L.reverse() – reverse *IN PLACE*

AttributedBlock.**sort**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

AttributedBlock.**__add__**()
x.__add__(y) <==> x+y

AttributedBlock.**__contains__**()
x.__contains__(y) <==> y in x

AttributedBlock.**__delitem__**()
x.__delitem__(y) <==> del x[y]

AttributedBlock.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

AttributedBlock.**__eq__**()
x.__eq__(y) <==> x==y

`AttributedBlock.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`AttributedBlock.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`AttributedBlock.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`AttributedBlock.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`AttributedBlock.`**`__iadd__`**`()`
x.__iadd__(y) <==> x+=y

`AttributedBlock.`**`__imul__`**`()`
x.__imul__(y) <==> x*=y

`AttributedBlock.`**`__iter__`**`()` <==> *iter(x)*

`AttributedBlock.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`AttributedBlock.`**`__len__`**`()` <==> *len(x)*

`AttributedBlock.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`AttributedBlock.`**`__mul__`**`()`
x.__mul__(n) <==> x*n

`AttributedBlock.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`AttributedBlock.`**`__repr__`**`()`

`AttributedBlock.`**`__reversed__`**`()`
L.__reversed__() – return a reverse iterator over the list

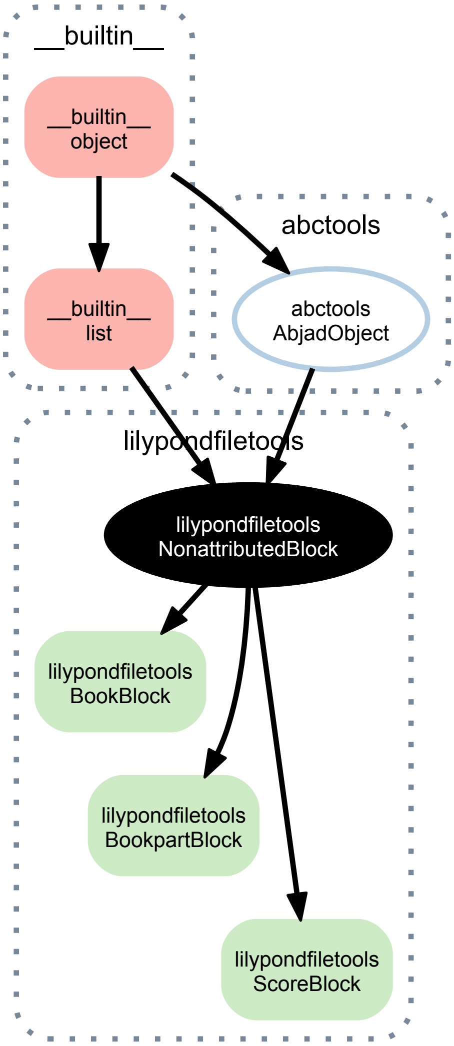
`AttributedBlock.`**`__rmul__`**`()`
x.__rmul__(n) <==> n*x

`AttributedBlock.`**`__setitem__`**`()`
x.__setitem__(i, y) <==> x[i]=y

`AttributedBlock.`**`__setslice__`**`()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.1.2 lilypondfiletools.NonattributedBlock



class `lilypondfiletools.`**`NonattributedBlock`**

New in version 2.0. Abjad model of LilyPond input file block with no attributes.

Read-only properties

`NonattributedBlock.`**`lilypond_format`**

`NonattributedBlock.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`NonattributedBlock.`**`is_formatted_when_empty`**

Methods

`NonattributedBlock.`**`append`**`()`

L.append(object) – append object to end

`NonattributedBlock.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`NonattributedBlock.`**`extend`**`()`
L.extend(iterable) – extend list by appending elements from the iterable

`NonattributedBlock.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`NonattributedBlock.`**`insert`**`()`
L.insert(index, object) – insert object before index

`NonattributedBlock.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`NonattributedBlock.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`NonattributedBlock.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`NonattributedBlock.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`NonattributedBlock.`**`__add__`**`()`
x.__add__(y) <==> x+y

`NonattributedBlock.`**`__contains__`**`()`
x.__contains__(y) <==> y in x

`NonattributedBlock.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`NonattributedBlock.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`NonattributedBlock.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`NonattributedBlock.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`NonattributedBlock.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`NonattributedBlock.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`NonattributedBlock.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`NonattributedBlock.`**`__iadd__`**`()`
x.__iadd__(y) <==> x+=y

`NonattributedBlock.`**`__imul__`**`()`
x.__imul__(y) <==> x*=y

`NonattributedBlock.`**`__iter__`**`()` <==> *iter(x)*

`NonattributedBlock.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`NonattributedBlock.`**`__len__`**`()` <==> *len(x)*

NonattributedBlock.**__lt__**()
x.__lt__(y) <==> x<y

NonattributedBlock.**__mul__**()
x.__mul__(n) <==> x*n

NonattributedBlock.**__ne__**()
x.__ne__(y) <==> x!=y

NonattributedBlock.**__repr__**()

NonattributedBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

NonattributedBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

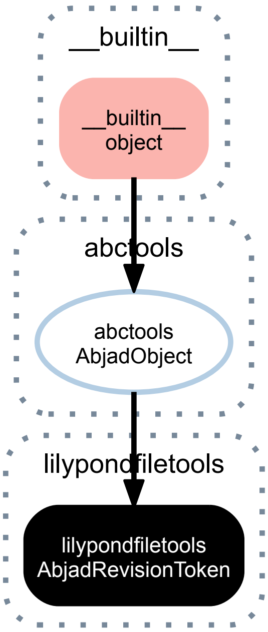
NonattributedBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

NonattributedBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2 Concrete Classes

14.2.1 lilypondfiletools.AbjadRevisionToken



class lilypondfiletools.**AbjadRevisionToken**
New in version 2.0. Abjad version token:

```
>>> lilypondfiletools.AbjadRevisionToken()
AbjadRevisionToken(Abjad revision ...)
```

Return Abjad version token.

Read-only properties

AbjadRevisionToken.**lilypond_format**
Format contribution of Abjad version token:

```
>>> lilypondfiletools.AbjadRevisionToken().lilypond_format
'Abjad revision ...'
```

Return string.

AbjadRevisionToken.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

AbjadRevisionToken.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

AbjadRevisionToken.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadRevisionToken.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjadRevisionToken.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadRevisionToken.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

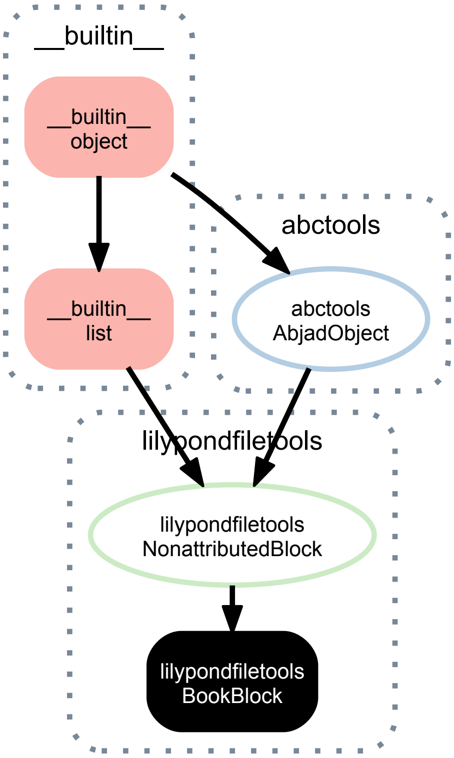
Raise exception.

AbjadRevisionToken.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjadRevisionToken.**__repr__**()

14.2.2 lilypondfiletools.BookBlock



class lilypondfiletools.**BookBlock**

New in version 2.0. Abjad model of LilyPond input file book block:

```
>>> book_block = lilypondfiletools.BookBlock()
```

```
>>> book_block
BookBlock()
```

```
>>> f(book_block)
\book {}
```

Return book block.

Read-only properties

BookBlock.**lilypond_format**

BookBlock.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

BookBlock.**is_formatted_when_empty**

Methods

BookBlock.**append**()

L.append(object) – append object to end

BookBlock.**count**(*value*) → integer – return number of occurrences of value

`BookBlock.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`BookBlock.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`BookBlock.insert()`
L.insert(index, object) – insert object before index

`BookBlock.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`BookBlock.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`BookBlock.reverse()`
L.reverse() – reverse *IN PLACE*

`BookBlock.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`BookBlock.__add__()`
x.__add__(y) <==> x+y

`BookBlock.__contains__()`
x.__contains__(y) <==> y in x

`BookBlock.__delitem__()`
x.__delitem__(y) <==> del x[y]

`BookBlock.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`BookBlock.__eq__()`
x.__eq__(y) <==> x==y

`BookBlock.__ge__()`
x.__ge__(y) <==> x>=y

`BookBlock.__getitem__()`
x.__getitem__(y) <==> x[y]

`BookBlock.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`BookBlock.__gt__()`
x.__gt__(y) <==> x>y

`BookBlock.__iadd__()`
x.__iadd__(y) <==> x+=y

`BookBlock.__imul__()`
x.__imul__(y) <==> x*=y

`BookBlock.__iter__()` <==> *iter(x)*

`BookBlock.__le__()`
x.__le__(y) <==> x<=y

`BookBlock.__len__()` <==> *len(x)*

BookBlock.**__lt__**()
x.__lt__(y) <==> x<y

BookBlock.**__mul__**()
x.__mul__(n) <==> x*n

BookBlock.**__ne__**()
x.__ne__(y) <==> x!=y

BookBlock.**__repr__**()

BookBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

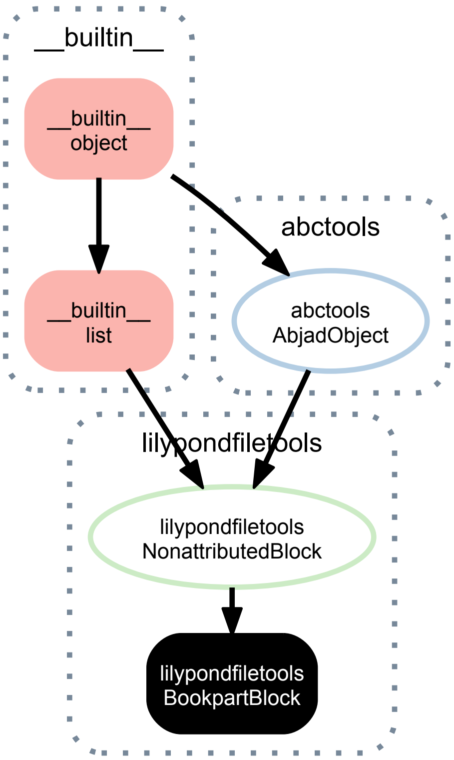
BookBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

BookBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

BookBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.3 lilypondfiletools.BookpartBlock



class lilypondfiletools.**BookpartBlock**
New in version 2.0. Abjad model of LilyPond input file bookpart block:

```
>>> bookpart_block = lilypondfiletools.BookpartBlock()
```

```
>>> bookpart_block
BookpartBlock()
```

```
>>> f(bookpart_block)
\bookpart {}
```

Return bookpart block.

Read-only properties

`BookpartBlock.`**`lilypond_format`**

`BookpartBlock.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`BookpartBlock.`**`is_formatted_when_empty`**

Methods

`BookpartBlock.`**`append`**`()`
L.append(object) – append object to end

`BookpartBlock.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`BookpartBlock.`**`extend`**`()`
L.extend(iterable) – extend list by appending elements from the iterable

`BookpartBlock.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`BookpartBlock.`**`insert`**`()`
L.insert(index, object) – insert object before index

`BookpartBlock.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`BookpartBlock.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`BookpartBlock.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`BookpartBlock.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`BookpartBlock.`**`__add__`**`()`
x.__add__(y) <==> x+y

`BookpartBlock.`**`__contains__`**`()`
x.__contains__(y) <==> y in x

`BookpartBlock.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`BookpartBlock.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`BookpartBlock.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

BookpartBlock.**__ge__**()
x.__ge__(y) <==> x>=y

BookpartBlock.**__getitem__**()
x.__getitem__(y) <==> x[y]

BookpartBlock.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

BookpartBlock.**__gt__**()
x.__gt__(y) <==> x>y

BookpartBlock.**__iadd__**()
x.__iadd__(y) <==> x+=y

BookpartBlock.**__imul__**()
x.__imul__(y) <==> x*=y

BookpartBlock.**__iter__**() <==> *iter(x)*

BookpartBlock.**__le__**()
x.__le__(y) <==> x<=y

BookpartBlock.**__len__**() <==> *len(x)*

BookpartBlock.**__lt__**()
x.__lt__(y) <==> x<y

BookpartBlock.**__mul__**()
x.__mul__(n) <==> x*n

BookpartBlock.**__ne__**()
x.__ne__(y) <==> x!=y

BookpartBlock.**__repr__**()

BookpartBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

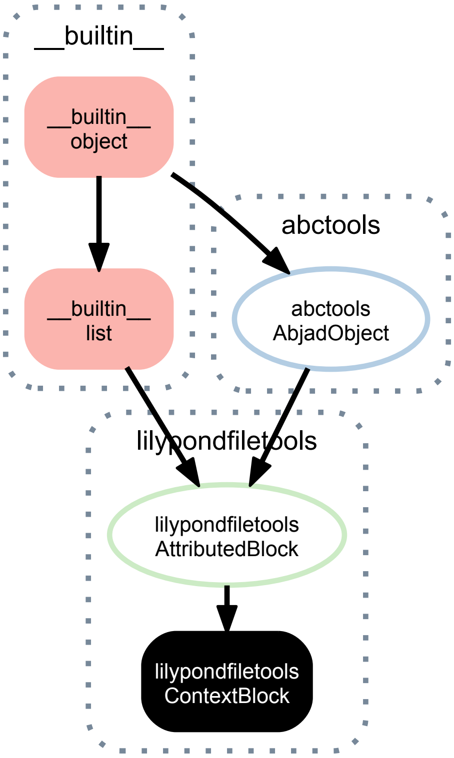
BookpartBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

BookpartBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

BookpartBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.4 lilypondfiletools.ContextBlock



class `lilypondfiletools.`**`ContextBlock`**(*context_name=None*)

New in version 2.5. Abjad model of LilyPond input file context block:

```
>>> context_block = lilypondfiletools.ContextBlock()
```

```
>>> context_block
ContextBlock()
```

```
>>> context_block.context_name = 'Score'
>>> context_block.override.bar_number.transparent = True
>>> scheme = schemetools.Scheme('end-of-line-invisible')
>>> context_block.override.time_signature.break_visibility = scheme
>>> context_block.set.proportionalNotationDuration = schemetools.SchemeMoment((1, 45))
```

```
>>> f(context_block)
\context {
    \Score
    \override BarNumber #'transparent = ##t
    \override TimeSignature #'break-visibility = #end-of-line-invisible
    proportionalNotationDuration = #(ly:make-moment 1 45)
}
```

Return context block.

Read-only properties

`ContextBlock.`**`accepts`**

`ContextBlock.`**`engraver_consists`**

`ContextBlock.`**`engraver_removals`**

`ContextBlock.`**`lilypond_format`**

`ContextBlock.`**`override`**

Read-only reference to LilyPond grob override component plug-in.

`ContextBlock.set`
Read-only reference LilyPond context setting component plug-in.

`ContextBlock.storage_format`
Storage format of Abjad object.

Return string.

Read/write properties

`ContextBlock.alias`
Read / write alias.

`ContextBlock.context_name`
Read / write context name.

`ContextBlock.is_formatted_when_empty`

`ContextBlock.name`
Read / write name.

`ContextBlock.type`
Read / write type.

Methods

`ContextBlock.append()`
L.append(object) – append object to end

`ContextBlock.count(value)` → integer – return number of occurrences of value

`ContextBlock.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`ContextBlock.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`ContextBlock.insert()`
L.insert(index, object) – insert object before index

`ContextBlock.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`ContextBlock.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`ContextBlock.reverse()`
L.reverse() – reverse *IN PLACE*

`ContextBlock.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`ContextBlock.__add__()`
x.__add__(y) <==> x+y

`ContextBlock.__contains__()`
x.__contains__(y) <==> y in x

`ContextBlock.__delitem__()`
x.__delitem__(y) <==> del x[y]

`ContextBlock.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`ContextBlock.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`ContextBlock.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`ContextBlock.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`ContextBlock.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ContextBlock.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`ContextBlock.`**`__iadd__`**`()`
x.__iadd__(y) <==> x+=y

`ContextBlock.`**`__imul__`**`()`
x.__imul__(y) <==> x*=y

`ContextBlock.`**`__iter__`**`()` <==> *iter(x)*

`ContextBlock.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`ContextBlock.`**`__len__`**`()` <==> *len(x)*

`ContextBlock.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`ContextBlock.`**`__mul__`**`()`
x.__mul__(n) <==> x*n

`ContextBlock.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`ContextBlock.`**`__repr__`**`()`

`ContextBlock.`**`__reversed__`**`()`
L.__reversed__() – return a reverse iterator over the list

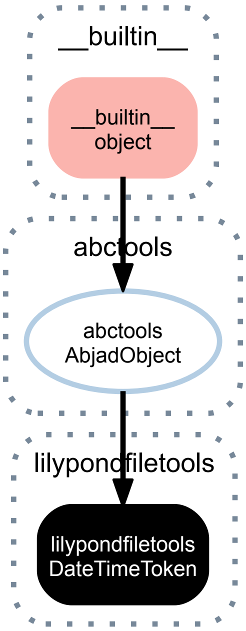
`ContextBlock.`**`__rmul__`**`()`
x.__rmul__(n) <==> n*x

`ContextBlock.`**`__setitem__`**`()`
x.__setitem__(i, y) <==> x[i]=y

`ContextBlock.`**`__setslice__`**`()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.5 lilypondfiletools.DateTimeToken



class lilypondfiletools.**DateTimeToken**

New in version 2.0. Date time token:

```
>>> lilypondfiletools.DateTimeToken()
DateTimeToken(...)
```

Return date / time token.

Read-only properties

DateTimeToken.**lilypond_format**

Format contribution of date time token:

```
>>> lilypondfiletools.DateTimeToken().lilypond_format
'...'
```

Return string.

DateTimeToken.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

DateTimeToken.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

DateTimeToken.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

DateTimeToken.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

DateTimeToken.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DateTimeToken.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

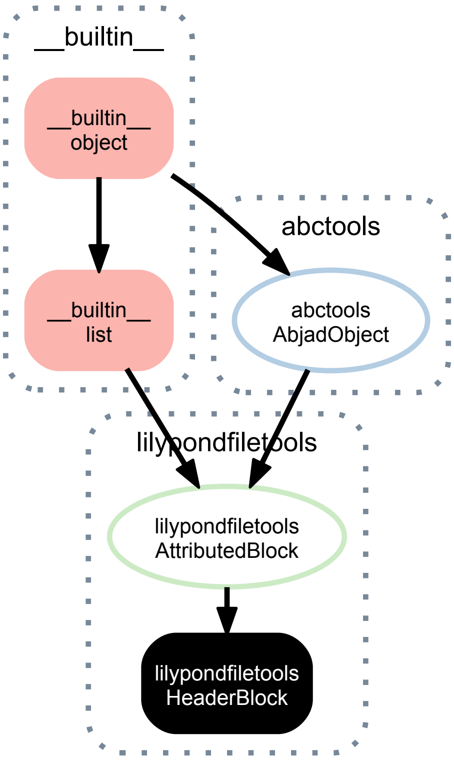
Raise exception.

DateTimeToken.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DateTimeToken.**__repr__**()

14.2.6 lilypondfiletools.HeaderBlock



class lilypondfiletools.**HeaderBlock**
New in version 2.0. Abjad model of LilyPond input file header block:

```
>>> header_block = lilypondfiletools.HeaderBlock()
```

```
>>> header_block
HeaderBlock()
```

```
>>> header_block.composer = markuptools.Markup('Josquin')
>>> header_block.title = markuptools.Markup('Missa sexti tonus')
```

```
>>> f(header_block)
\header {
    composer = \markup { Josquin }
    title = \markup { Missa sexti tonus }
}
```

Return header block.

Read-only properties

`HeaderBlock.lilypond_format`

`HeaderBlock.storage_format`
Storage format of Abjad object.

Return string.

Read/write properties

`HeaderBlock.is_formatted_when_empty`

Methods

`HeaderBlock.append()`
L.append(object) – append object to end

`HeaderBlock.count(value)` → integer – return number of occurrences of value

`HeaderBlock.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`HeaderBlock.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`HeaderBlock.insert()`
L.insert(index, object) – insert object before index

`HeaderBlock.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`HeaderBlock.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`HeaderBlock.reverse()`
L.reverse() – reverse *IN PLACE*

`HeaderBlock.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`HeaderBlock.__add__()`
x.__add__(y) <==> x+y

`HeaderBlock.__contains__()`
x.__contains__(y) <==> y in x

`HeaderBlock.__delitem__()`
x.__delitem__(y) <==> del x[y]

`HeaderBlock.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`HeaderBlock.__eq__()`
x.__eq__(y) <==> x==y

`HeaderBlock.__ge__()`
x.__ge__(y) <==> x>=y

`HeaderBlock.__getitem__()`
x.__getitem__(y) <==> x[y]

`HeaderBlock.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`HeaderBlock.__gt__()`
x.__gt__(y) <==> x>y

`HeaderBlock.__iadd__()`
x.__iadd__(y) <==> x+=y

`HeaderBlock.__imul__()`
x.__imul__(y) <==> x*=y

`HeaderBlock.__iter__()` <==> *iter(x)*

`HeaderBlock.__le__()`
x.__le__(y) <==> x<=y

`HeaderBlock.__len__()` <==> *len(x)*

`HeaderBlock.__lt__()`
x.__lt__(y) <==> x<y

`HeaderBlock.__mul__()`
x.__mul__(n) <==> x*n

`HeaderBlock.__ne__()`
x.__ne__(y) <==> x!=y

`HeaderBlock.__repr__()`

`HeaderBlock.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

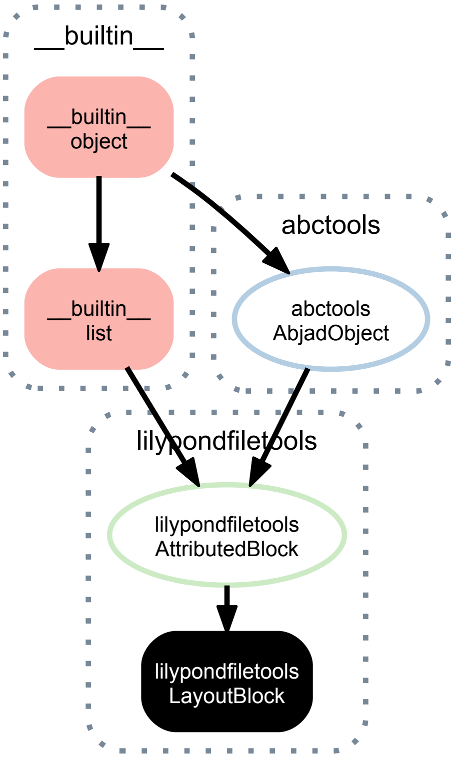
`HeaderBlock.__rmul__()`
x.__rmul__(n) <==> n*x

`HeaderBlock.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`HeaderBlock.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.7 lilypondfiletools.LayoutBlock



class `lilypondfiletools.LayoutBlock`

New in version 2.0. Abjad model of LilyPond input file layout block:

```
>>> layout_block = lilypondfiletools.LayoutBlock()
```

```
>>> layout_block
LayoutBlock()
```

```
>>> layout_block.indent = 0
>>> layout_block.ragged_right = True
```

```
>>> f(layout_block)
\layout {
    indent = #0
    ragged-right = ##t
}
```

Return layout block.

Read-only properties

`LayoutBlock.context_blocks`

Read-only list of context blocks:

```
>>> layout_block = lilypondfiletools.LayoutBlock()
```

```
>>> context_block = lilypondfiletools.ContextBlock('Score')
>>> context_block.override.bar_number.transparent = True
```

```
>>> scheme_expr = schemetools.Scheme('end-of-line-invisible')
>>> context_block.override.time_signature.break_visibility = scheme_expr
>>> layout_block.context_blocks.append(context_block)
```

```
>>> f(layout_block)
\layout {
```

```
        \context {
            \Score
            \override BarNumber #'transparent = ##t
            \override TimeSignature #'break-visibility = #end-of-line-invisible
        }
    }
```

Return list.

`LayoutBlock.`**`contexts`**
DEPRECATED. USE CONTEXT_BLOCKS INSTEAD.

`LayoutBlock.`**`lilypond_format`**

`LayoutBlock.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`LayoutBlock.`**`is_formatted_when_empty`**

Methods

`LayoutBlock.`**`append`**`()`
L.append(object) – append object to end

`LayoutBlock.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`LayoutBlock.`**`extend`**`()`
L.extend(iterable) – extend list by appending elements from the iterable

`LayoutBlock.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`LayoutBlock.`**`insert`**`()`
L.insert(index, object) – insert object before index

`LayoutBlock.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`LayoutBlock.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`LayoutBlock.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`LayoutBlock.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`LayoutBlock.`**`__add__`**`()`
x.__add__(y) <==> x+y

`LayoutBlock.`**`__contains__`**`()`
x.__contains__(y) <==> y in x

`LayoutBlock.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

LayoutBlock.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

LayoutBlock.**__eq__**()
x.__eq__(y) <==> x==y

LayoutBlock.**__ge__**()
x.__ge__(y) <==> x>=y

LayoutBlock.**__getitem__**()
x.__getitem__(y) <==> x[y]

LayoutBlock.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

LayoutBlock.**__gt__**()
x.__gt__(y) <==> x>y

LayoutBlock.**__iadd__**()
x.__iadd__(y) <==> x+=y

LayoutBlock.**__imul__**()
x.__imul__(y) <==> x*=y

LayoutBlock.**__iter__**() <==> *iter(x)*

LayoutBlock.**__le__**()
x.__le__(y) <==> x<=y

LayoutBlock.**__len__**() <==> *len(x)*

LayoutBlock.**__lt__**()
x.__lt__(y) <==> x<y

LayoutBlock.**__mul__**()
x.__mul__(n) <==> x*n

LayoutBlock.**__ne__**()
x.__ne__(y) <==> x!=y

LayoutBlock.**__repr__**()

LayoutBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

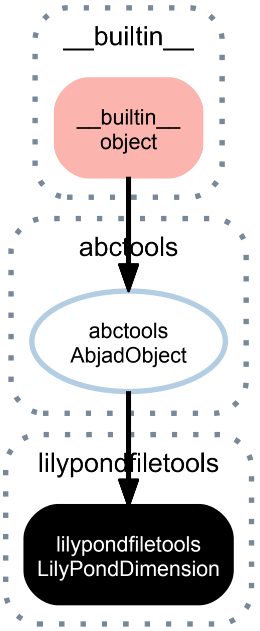
LayoutBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

LayoutBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

LayoutBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.8 lilypondfiletools.LilyPondDimension



class lilypondfiletools.**LilyPondDimension**(*value*, *unit*)
Abjad model of page dimensions in LilyPond:

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
```

```
>>> f(dimension)
2.0\in
```

Return LilyPondDimension instance.

Read-only properties

LilyPondDimension.**lilypond_format**

LilyPondDimension.**storage_format**
Storage format of Abjad object.

Return string.

LilyPondDimension.**unit**

LilyPondDimension.**value**

Special methods

LilyPondDimension.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

LilyPondDimension.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondDimension.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondDimension.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondDimension.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

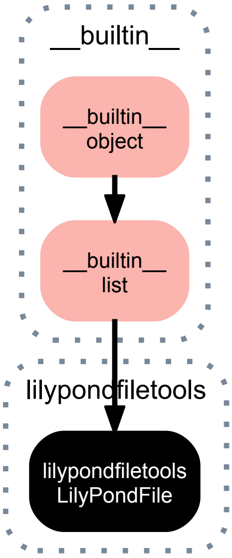
LilyPondDimension.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LilyPondDimension.**__repr__**()
Interpreter representation of Abjad object.

Return string.

14.2.9 lilypondfiletools.LilyPondFile



class lilypondfiletools.**LilyPondFile**
New in version 2.0. Abjad model of LilyPond input file:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
>>> lilypond_file.file_initial_user_comments.append('File construct as an example.')
>>> lilypond_file.file_initial_user_comments.append('Parts shown here for positioning.')
>>> lilypond_file.file_initial_user_includes.append('external-settings-file-1.ly')
>>> lilypond_file.file_initial_user_includes.append('external-settings-file-2.ly')
>>> lilypond_file.default_paper_size = 'letter', 'portrait'
>>> lilypond_file.global_staff_size = 16
>>> lilypond_file.header_block.composer = markuptools.Markup('Josquin')
>>> lilypond_file.header_block.title = markuptools.Markup('Missa sexti tonus')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.layout_block.left_margin = 15
>>> lilypond_file.paper_block.oddFooterMarkup = markuptools.Markup('The odd-page footer')
>>> lilypond_file.paper_block.evenFooterMarkup = markuptools.Markup('The even-page footer')
```

```
>>> f(lilypond_file)
% Abjad revision 3719
% 2010-09-24 09:01

% File construct as an example.
% Parts shown here for positioning.
```

```
\version "2.13.32"
\include "english.ly"
\include "/Users/trevorbaca/Documents/abjad/trunk/abjad/cfg/abjad.scm"

\include "external-settings-file-1.ly"
\include "external-settings-file-2.ly"

#(set-default-paper-size "letter" 'portrait)
#(set-global-staff-size 16)

\header {
    composer = \markup { Josquin }
    title = \markup { Missa sexti tonus }
}

\layout {
    indent = #0
    left-margin = #15
}

\paper {
    evenFooterMarkup = \markup { The even-page footer }
    oddFooterMarkup = \markup { The odd-page footer }
}

\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

Read-only properties

`LilyPondFile.`**`lilypond_format`**
Format-time contribution of LilyPond file.

Read/write properties

`LilyPondFile.`**`default_paper_size`**
LilyPond default paper size.

`LilyPondFile.`**`file_initial_system_comments`**
Read-only list of file-initial system comments.

`LilyPondFile.`**`file_initial_system_includes`**
List of file-initial system include commands.

`LilyPondFile.`**`file_initial_user_comments`**
Read-only list of file-initial user comments.

`LilyPondFile.`**`file_initial_user_includes`**
List of file-initial user include commands.

`LilyPondFile.`**`global_staff_size`**
LilyPond global staff size.

Methods

`LilyPondFile.`**`append`**`()`
L.append(object) – append object to end

`LilyPondFile.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`LilyPondFile.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`LilyPondFile.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`LilyPondFile.insert()`
L.insert(index, object) – insert object before index

`LilyPondFile.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`LilyPondFile.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`LilyPondFile.reverse()`
L.reverse() – reverse *IN PLACE*

`LilyPondFile.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`LilyPondFile.__add__()`
x.__add__(y) <==> x+y

`LilyPondFile.__contains__()`
x.__contains__(y) <==> y in x

`LilyPondFile.__delitem__()`
x.__delitem__(y) <==> del x[y]

`LilyPondFile.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`LilyPondFile.__eq__()`
x.__eq__(y) <==> x==y

`LilyPondFile.__ge__()`
x.__ge__(y) <==> x>=y

`LilyPondFile.__getitem__()`
x.__getitem__(y) <==> x[y]

`LilyPondFile.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`LilyPondFile.__gt__()`
x.__gt__(y) <==> x>y

`LilyPondFile.__iadd__()`
x.__iadd__(y) <==> x+=y

`LilyPondFile.__imul__()`
x.__imul__(y) <==> x*=y

`LilyPondFile.__iter__()` <==> *iter(x)*

`LilyPondFile.__le__()`
x.__le__(y) <==> x<=y

`LilyPondFile.__len__()` <==> *len(x)*

`LilyPondFile.__lt__()`
x.__lt__(y) <==> x<y

`LilyPondFile.__mul__()`
x.__mul__(n) <==> x*n

`LilyPondFile.__ne__()`
x.__ne__(y) <==> x!=y

`LilyPondFile.__repr__()`

`LilyPondFile.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

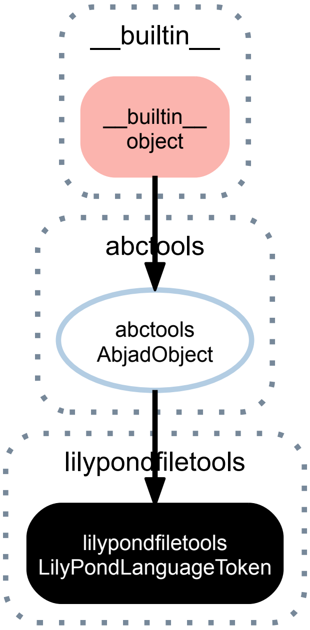
`LilyPondFile.__rmul__()`
x.__rmul__(n) <==> n*x

`LilyPondFile.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`LilyPondFile.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.10 lilypondfiletools.LilyPondLanguageToken



class `lilypondfiletools.LilyPondLanguageToken`
New in version 2.0. LilyPond language token:

```
>>> lilypondfiletools.LilyPondLanguageToken()
LilyPondLanguageToken('english')
```

Return LilyPond language token.

Read-only properties

`LilyPondLanguageToken.lilypond_format`
Format contribution of LilyPond language token:

```
>>> lilypondfiletools.LilyPondLanguageToken().lilypond_format
'\\language "english"'
```

Return string.

LilyPondLanguageToken.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

LilyPondLanguageToken.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

LilyPondLanguageToken.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LilyPondLanguageToken.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

LilyPondLanguageToken.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LilyPondLanguageToken.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

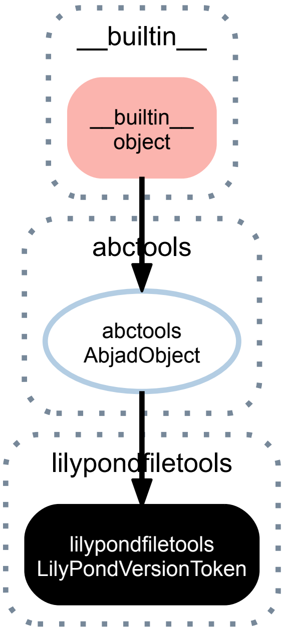
LilyPondLanguageToken.**__ne__**(*expr*)

Defined equal to the opposite of equality.

Return boolean.

LilyPondLanguageToken.**__repr__**()

14.2.11 lilypondfiletools.LilyPondVersionToken



class `lilypondfiletools.`**`LilyPondVersionToken`**

New in version 2.0. LilyPond version token:

```
>>> lilypondfiletools.LilyPondVersionToken()
LilyPondVersionToken(\version "...")
```

Return LilyPond version token.

Read-only properties

`LilyPondVersionToken.`**`lilypond_format`**

Format contribution of LilyPond version token:

```
>>> lilypondfiletools.LilyPondVersionToken().lilypond_format
'\\version "..."'
```

Return string.

`LilyPondVersionToken.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`LilyPondVersionToken.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`LilyPondVersionToken.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`LilyPondVersionToken.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

LilyPondVersionToken.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondVersionToken.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

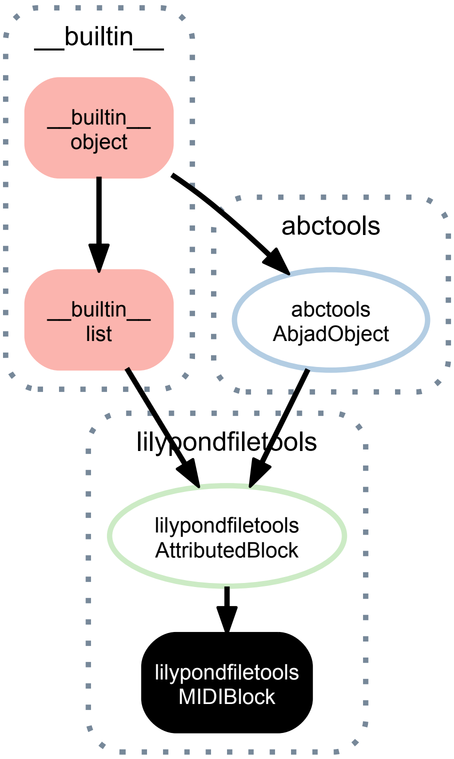
Raise exception.

LilyPondVersionToken.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LilyPondVersionToken.**__repr__**()

14.2.12 lilypondfiletools.MIDIBlock



class lilypondfiletools.**MIDIBlock**
New in version 2.0. Abjad model of LilyPond input file MIDI block:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> score = Score([staff])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
```

```
>>> lilypond_file.score_block.append(lilypondfiletools.MIDIBlock())
```

```
>>> layout_block = lilypondfiletools.LayoutBlock()
>>> layout_block.is_formatted_when_empty = True
>>> lilypond_file.score_block.append(layout_block)
```

```
>>> f(lilypond_file.score_block)
\score {
    \new Score <<
        \new Staff {
            c'4
            d'4
            e'4
```

```
                f'4
            }
        >>
        \midi {}
        \layout {}
    }
```

MIDI blocks are formatted even when they are empty.

The example here appends MIDI and layout blocks to a score block. Doing this allows LilyPond to create both MIDI and PDF output from a single input file.

Read the LilyPond docs on LilyPond file structure for the details as to why this is the case.

Read-only properties

MIDIBlock.**lilypond_format**

MIDIBlock.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

MIDIBlock.**is_formatted_when_empty**

Methods

MIDIBlock.**append**()
L.append(object) – append object to end

MIDIBlock.**count**(*value*) → integer – return number of occurrences of value

MIDIBlock.**extend**()
L.extend(iterable) – extend list by appending elements from the iterable

MIDIBlock.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

MIDIBlock.**insert**()
L.insert(index, object) – insert object before index

MIDIBlock.**pop**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

MIDIBlock.**remove**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

MIDIBlock.**reverse**()
L.reverse() – reverse *IN PLACE*

MIDIBlock.**sort**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

MIDIBlock.**__add__**()
x.__add__(y) <==> x+y

MIDIBlock.**__contains__**()
x.__contains__(y) <==> y in x

MIDIBlock.**__delitem__**()
x.__delitem__(y) <==> del x[y]

MIDIBlock.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

MIDIBlock.**__eq__**()
x.__eq__(y) <==> x==y

MIDIBlock.**__ge__**()
x.__ge__(y) <==> x>=y

MIDIBlock.**__getitem__**()
x.__getitem__(y) <==> x[y]

MIDIBlock.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

MIDIBlock.**__gt__**()
x.__gt__(y) <==> x>y

MIDIBlock.**__iadd__**()
x.__iadd__(y) <==> x+=y

MIDIBlock.**__imul__**()
x.__imul__(y) <==> x*=y

MIDIBlock.**__iter__**() <==> *iter(x)*

MIDIBlock.**__le__**()
x.__le__(y) <==> x<=y

MIDIBlock.**__len__**() <==> *len(x)*

MIDIBlock.**__lt__**()
x.__lt__(y) <==> x<y

MIDIBlock.**__mul__**()
x.__mul__(n) <==> x*n

MIDIBlock.**__ne__**()
x.__ne__(y) <==> x!=y

MIDIBlock.**__repr__**()

MIDIBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

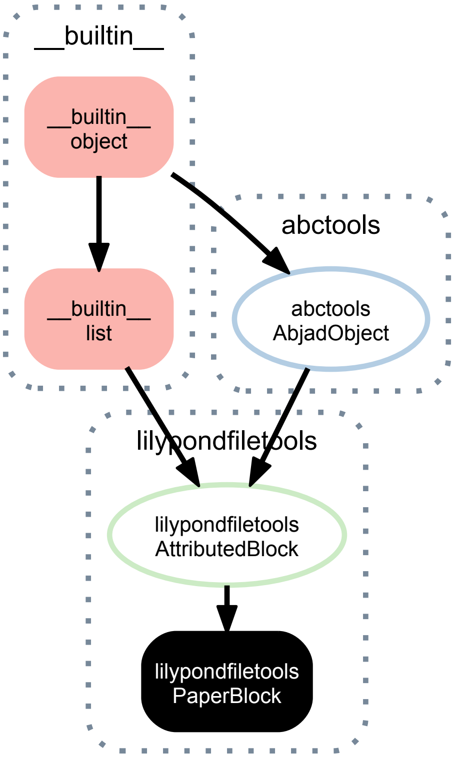
MIDIBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

MIDIBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

MIDIBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.13 lilypondfiletools.PaperBlock



class lilypondfiletools.**PaperBlock**

New in version 2.0. Abjad model of LilyPond input file paper block:

```
>>> paper_block = lilypondfiletools.PaperBlock()
```

```
>>> paper_block
PaperBlock()
```

```
>>> paper_block.print_page_number = True
>>> paper_block.print_first_page_number = False
```

```
>>> f(paper_block)
\paper {
    print-first-page-number = ##f
    print-page-number = ##t
}
```

Return paper block.

Read-only properties

PaperBlock.**lilypond_format**

PaperBlock.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

PaperBlock.**is_formatted_when_empty**

PaperBlock.**minimal_page_breaking**

Methods

`PaperBlock.append()`
L.append(object) – append object to end

`PaperBlock.count(value)` → integer – return number of occurrences of value

`PaperBlock.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`PaperBlock.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`PaperBlock.insert()`
L.insert(index, object) – insert object before index

`PaperBlock.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`PaperBlock.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`PaperBlock.reverse()`
L.reverse() – reverse *IN PLACE*

`PaperBlock.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`PaperBlock.__add__()`
x.__add__(y) <==> x+y

`PaperBlock.__contains__()`
x.__contains__(y) <==> y in x

`PaperBlock.__delitem__()`
x.__delitem__(y) <==> del x[y]

`PaperBlock.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`PaperBlock.__eq__()`
x.__eq__(y) <==> x==y

`PaperBlock.__ge__()`
x.__ge__(y) <==> x>=y

`PaperBlock.__getitem__()`
x.__getitem__(y) <==> x[y]

`PaperBlock.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`PaperBlock.__gt__()`
x.__gt__(y) <==> x>y

`PaperBlock.__iadd__()`
x.__iadd__(y) <==> x+=y

`PaperBlock.__imul__()`
x.__imul__(y) <==> x*=y

`PaperBlock.__iter__()` <==> *iter(x)*

PaperBlock.**__le__**()
x.__le__(y) <==> x<=y

PaperBlock.**__len__**() <==> *len(x)*

PaperBlock.**__lt__**()
x.__lt__(y) <==> x<y

PaperBlock.**__mul__**()
x.__mul__(n) <==> x*n

PaperBlock.**__ne__**()
x.__ne__(y) <==> x!=y

PaperBlock.**__repr__**()

PaperBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

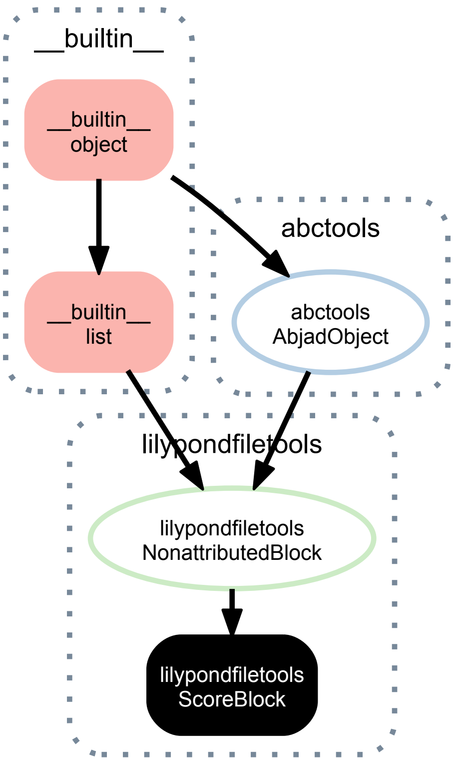
PaperBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

PaperBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

PaperBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.2.14 lilypondfiletools.ScoreBlock



class lilypondfiletools.**ScoreBlock**
New in version 2.0. Abjad model of LilyPond input file score block:

```
>>> score_block = lilypondfiletools.ScoreBlock()
```

```
>>> score_block
ScoreBlock()
```

```
>>> score_block.append(Staff([]))
>>> f(score_block)
\score {
    \new Staff {
    }
}
```

ScoreBlocks does not format when empty, as this generates a parser error in LilyPond:

```
>>> score_block = lilypondfiletools.ScoreBlock()
>>> score_block.lilypond_format == ''
True
```

Return score block.

Read-only properties

`ScoreBlock.`**`lilypond_format`**

`ScoreBlock.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`ScoreBlock.`**`is_formatted_when_empty`**

Methods

`ScoreBlock.`**`append`**`()`
L.append(object) – append object to end

`ScoreBlock.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`ScoreBlock.`**`extend`**`()`
L.extend(iterable) – extend list by appending elements from the iterable

`ScoreBlock.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`ScoreBlock.`**`insert`**`()`
L.insert(index, object) – insert object before index

`ScoreBlock.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`ScoreBlock.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`ScoreBlock.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`ScoreBlock.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

ScoreBlock.**__add__**()
x.__add__(y) <==> x+y

ScoreBlock.**__contains__**()
x.__contains__(y) <==> y in x

ScoreBlock.**__delitem__**()
x.__delitem__(y) <==> del x[y]

ScoreBlock.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

ScoreBlock.**__eq__**()
x.__eq__(y) <==> x==y

ScoreBlock.**__ge__**()
x.__ge__(y) <==> x>=y

ScoreBlock.**__getitem__**()
x.__getitem__(y) <==> x[y]

ScoreBlock.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ScoreBlock.**__gt__**()
x.__gt__(y) <==> x>y

ScoreBlock.**__iadd__**()
x.__iadd__(y) <==> x+=y

ScoreBlock.**__imul__**()
x.__imul__(y) <==> x*=y

ScoreBlock.**__iter__**() <==> *iter(x)*

ScoreBlock.**__le__**()
x.__le__(y) <==> x<=y

ScoreBlock.**__len__**() <==> *len(x)*

ScoreBlock.**__lt__**()
x.__lt__(y) <==> x<y

ScoreBlock.**__mul__**()
x.__mul__(n) <==> x*n

ScoreBlock.**__ne__**()
x.__ne__(y) <==> x!=y

ScoreBlock.**__repr__**()

ScoreBlock.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

ScoreBlock.**__rmul__**()
x.__rmul__(n) <==> n*x

ScoreBlock.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

ScoreBlock.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

14.3 Functions

14.3.1 lilypondfiletools.make_basic_lilypond_file

lilypondfiletools.**make_basic_lilypond_file**(*music=None*)

New in version 2.0. Make basic LilyPond file with *music*:

```
>>> score = Score([Staff("c'8 d'8 e'8 f'8")])
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
>>> lilypond_file.header_block.composer = markuptools.Markup('Josquin')
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.paper_block.top_margin = 15
>>> lilypond_file.paper_block.left_margin = 15
```

```
>>> f(lilypond_file)
\header {
    composer = \markup { Josquin }
}

\layout {
    indent = #0
}

\paper {
    left-margin = #15
    top-margin = #15
}

\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
>>
```

Equip LilyPond file with header, layout and paper blocks.

Return LilyPond file.

14.3.2 lilypondfiletools.make_floating_time_signature_lilypond_file

lilypondfiletools.**make_floating_time_signature_lilypond_file**(*music=None*)

New in version 2.10. Make floating time signature LilyPond file from *music*.

Function creates a basic LilyPond file.

Function then applies many layout settings.

View source here for the complete inventory of settings applied.

Returns LilyPond file object.

14.3.3 lilypondfiletools.make_time_signature_context_block

lilypondfiletools.**make_time_signature_context_block**(*font_size=3*, *padding=4*)

New in version 2.9. Make time signature context block:

```
>>> context_block = lilypondfiletools.make_time_signature_context_block()
```

```
>>> f(context_block)
\context {
    \type Engraver_group
    \name TimeSignatureContext
    \consists Axis_group_engraver
```

```
        \consists Time_signature_engraver
        \override TimeSignature #'X-extent = #'(0 . 0)
        \override TimeSignature #'X-offset = #ly:self-alignment-interface::x-aligned-on-self
        \override TimeSignature #'Y-extent = #'(0 . 0)
        \override TimeSignature #'break-align-symbol = ##f
        \override TimeSignature #'break-visibility = #end-of-line-invisible
        \override TimeSignature #'font-size = #3
        \override TimeSignature #'self-alignment-X = #center
        \override VerticalAxisGroup #'default-staff-staff-spacing = #'(
            (basic-distance . 0) (minimum-distance . 0) (padding . 4) (stretchability . 0))
    }
```

Return context block.

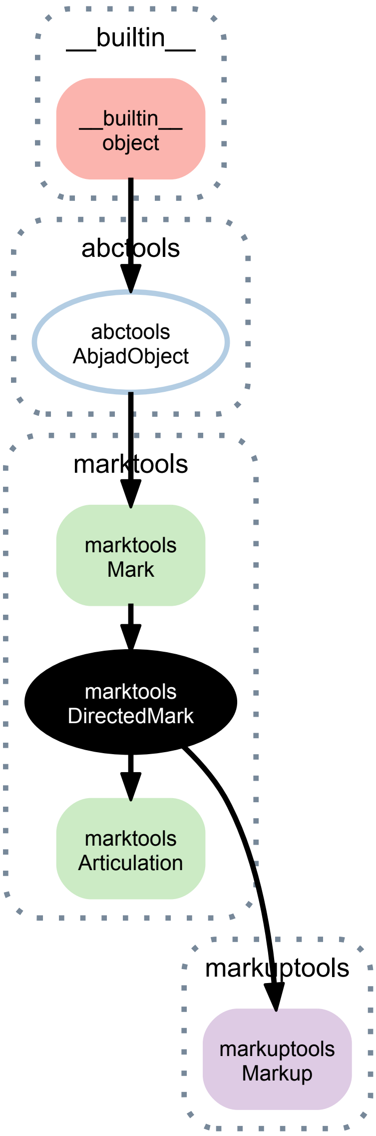
CHAPTER

FIFTEEN

MARKTOOLS

15.1 Abstract Classes

15.1.1 marktools.DirectedMark



class `marktools.DirectedMark`(**args*, ***kwargs*)

Abstract base class of Marks which possess a vertical, typographic direction, i.e. above or below the staff.

Read-only properties

DirectedMark.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

DirectedMark.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

DirectedMark.**direction**

Methods

DirectedMark.**attach**(*start_component*)

Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

DirectedMark.**detach**()

Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

DirectedMark.**__call__**(**args*)

DirectedMark.**__copy__**(**args*)

DirectedMark.**__deepcopy__**(**args*)

DirectedMark.**__delattr__**(**args*)

DirectedMark.**__eq__**(*arg*)

DirectedMark.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectedMark.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DirectedMark.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectedMark.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

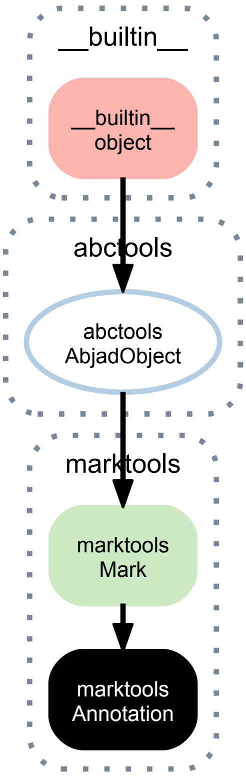
Raise exception.

DirectedMark.**__ne__**(*arg*)

DirectedMark.**__repr__**()

15.2 Concrete Classes

15.2.1 marktools.Annotation



class marktools.**Annotation**(**args*)
New in version 2.0. User-defined annotation:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> pitch = pitchtools.NamedChromaticPitch('ds')
>>> marktools.Annotation('special pitch', pitch)(staff[0])
Annotation('special pitch', NamedChromaticPitch('ds'))(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

Annotations contribute no formatting.

Annotations implement `__slots__`.

Read-only properties

`Annotation.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Annotation.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`Annotation.`**`name`**
Get name of annotation:

```
>>> pitch = pitchtools.NamedChromaticPitch('ds')
>>> annotation = marktools.Annotation('special_pitch', pitch)
>>> annotation.name
'special_pitch'
```

Set name of annotation:

```
>>> annotation.name = 'revised special pitch'
>>> annotation.name
'revised special pitch'
```

Set string.

`Annotation.`**`value`**
Get value of annotation:

```
>>> annotation.value
NamedChromaticPitch('ds')
```

Set value of annotation:

```
>>> annotation.value = pitchtools.NamedChromaticPitch('e')
>>> annotation.value
NamedChromaticPitch('e')
```

Set arbitrary object.

Methods

`Annotation.`**`attach`**`(`*start_component*`)`
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

`Annotation.`**`detach`**`()`
Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

`Annotation.`**`__call__`**`(`**args*`)`

`Annotation.`**`__copy__`**`(`**args*`)`

`Annotation.`**`__deepcopy__`**`(`**args*`)`

`Annotation.`**`__delattr__`**`(`**args*`)`

`Annotation.`**`__eq__`**`(`*arg*`)`

`Annotation.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Annotation.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Annotation.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

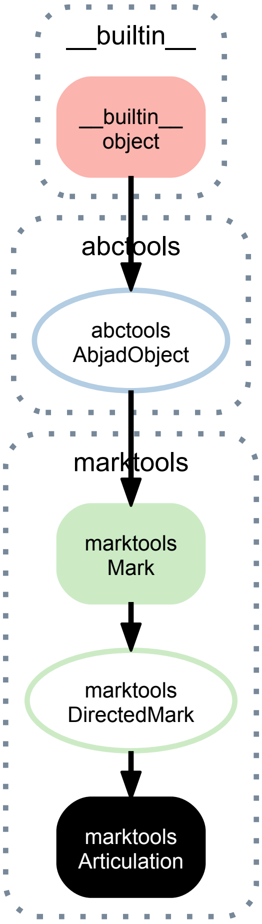
Annotation.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Annotation.**__ne__**(*arg*)

Annotation.**__repr__**()

15.2.2 marktools.Articulation



class marktools.**Articulation**(**args*)
Abjad model of musical articulation.

Initialize from articulation name:

```
>>> marktools.Articulation('staccato')
Articulation('staccato')
```

Initialize from articulation abbreviation:

```
>>> marktools.Articulation('.')
Articulation('.')
```

Initialize from other articulation:

```
>>> articulation = marktools.Articulation('staccato')
>>> marktools.Articulation(articulation)
Articulation('staccato')
```

Initialize with direction:

```
>>> marktools.Articulation('staccato', Up)
Articulation('staccato', Up)
```

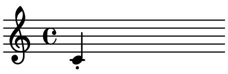
Attach to note, rest or chord after initialization:

```
>>> note = Note("c'4")
```

```
>>> marktools.Articulation('staccato')(note)
Articulation('staccato')(c'4)

>>> f(note)
c'4 -\staccato
```

```
>>> show(note)
```



Articulations implement `__slots__`.

Read-only properties

`Articulation.`**`lilypond_format`**
Read-only LilyPond format string of articulation:

```
>>> articulation = marktools.Articulation('marcato', Up)
>>> articulation.lilypond_format
'^\marcato'
```

Return string.

`Articulation.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Articulation.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`Articulation.`**`direction`**

`Articulation.`**`name`**
Get name of articulation:

```
>>> articulation = marktools.Articulation('staccato', Up)
>>> articulation.name
'staccato'
```

Set name of articulation:

```
>>> articulation.name = 'marcato'
>>> articulation.name
'marcato'
```

Set string.

Methods

`Articulation.`**`attach`**`(`*start_component*`)`

Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

`Articulation.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

`Articulation.`**`__call__`**`(`**args*`)`

`Articulation.`**`__copy__`**`(`**args*`)`

`Articulation.`**`__deepcopy__`**`(`**args*`)`

`Articulation.`**`__delattr__`**`(`**args*`)`

`Articulation.`**`__eq__`**`(`*expr*`)`

`Articulation.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`Articulation.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`Articulation.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`Articulation.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

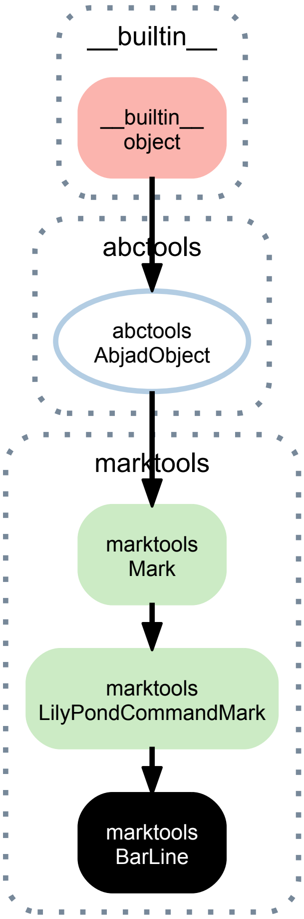
Raise exception.

`Articulation.`**`__ne__`**`(`*arg*`)`

`Articulation.`**`__repr__`**`()`

`Articulation.`**`__str__`**`()`

15.2.3 marktools.BarLine



class marktools.**BarLine**(*bar_line_string='|'*, *format_slot='after'*)

New in version 2.4. Abjad model of bar line:

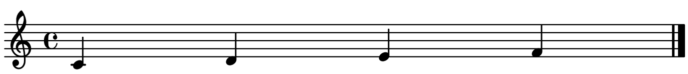
```
>>> staff = Staff("c'4 d'4 e'4 f'4")
```

```
>>> bar_line = marktools.BarLine('|.')(staff[-1])
```

```
>>> bar_line
BarLine('|.')(f'4)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    \bar "|."
}
```

```
>>> show(staff)
```



Return bar line.

Read-only properties

BarLine.**lilypond_format**

Read-only LilyPond input format of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')(note)
>>> lilypond_command.lilypond_format
'\slurDotted'
```

Return string.

BarLine.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

BarLine.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

BarLine.**bar_line_string**

Get bar line string of bar line:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> bar_line = marktools.BarLine()(staff[-1])
>>> bar_line.bar_line_string
'|'
```

Set bar line string of bar line:

```
>>> bar_line.bar_line_string = '|.'
>>> bar_line.bar_line_string
'|.'
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    \bar "|."
}
```

Set string.

BarLine.**command_name**

Get command name of LilyPond command mark:

```
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')
>>> lilypond_command.command_name
'slurDotted'
```

Set command name of LilyPond command mark:

```
>>> lilypond_command.command_name = 'slurDashed'
>>> lilypond_command.command_name
'slurDashed'
```

Set string.

`BarLine.`**`format_slot`**

New in version 2.3. Get format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('break', 'after')
>>> lilypond_command.format_slot
'after'
```

Set format slot of LiyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('break', 'after')
>>> lilypond_command.format_slot = 'before'
>>> lilypond_command.format_slot
'before'
```

Set to `'before'`, `'after'`, `'opening'`, `'closing'`, `'right'` or none.

Methods

`BarLine.`**`attach`**`(`*start_component*`)`

Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

`BarLine.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

`BarLine.`**`__call__`**`(`**args*`)`

`BarLine.`**`__copy__`**`(`**args*`)`

`BarLine.`**`__deepcopy__`**`(`**args*`)`

`BarLine.`**`__delattr__`**`(`**args*`)`

`BarLine.`**`__eq__`**`(`*arg*`)`

BarLine.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BarLine.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BarLine.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

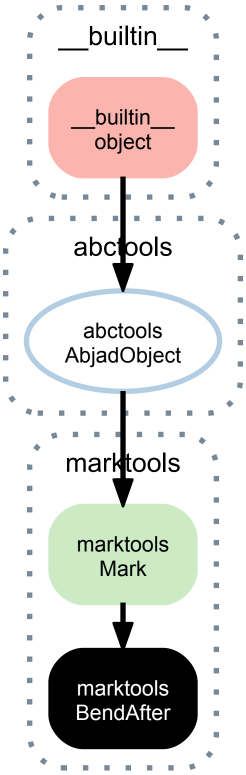
BarLine.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BarLine.**__ne__**(*arg*)

BarLine.**__repr__**()

15.2.4 marktools.BendAfter



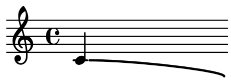
class marktools.**BendAfter**(**args*)
Abjad model of a fall or doit:

```
>>> note = Note("c'4")
```

```
>>> marktools.BendAfter(-4)(note)
BendAfter(-4.0)(c'4)
```

```
>>> f(note)
c'4 - \bendAfter #'-4.0
```

```
>>> show(note)
```



BendAfter implements `__slots__`.

Read-only properties

`BendAfter.`**`lilypond_format`**
Read-only LilyPond format string:

```
>>> bend = marktools.BendAfter(-4)
>>> bend.lilypond_format
"- \\bendAfter #'-4.0"
```

Return string.

`BendAfter.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`BendAfter.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`BendAfter.`**`bend_amount`**
Get bend amount:

```
>>> bend = marktools.BendAfter(8)
>>> bend.bend_amount
8.0
```

Set bend amount:

```
>>> bend.bend_amount = -4
>>> bend.bend_amount
-4.0
```

Set float.

Methods

`BendAfter.`**`attach`**(*start_component*)
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

BendAfter.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

BendAfter.**__call__**(**args*)

BendAfter.**__copy__**(**args*)

BendAfter.**__deepcopy__**(**args*)

BendAfter.**__delattr__**(**args*)

BendAfter.**__eq__**(*expr*)

BendAfter.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BendAfter.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BendAfter.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BendAfter.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

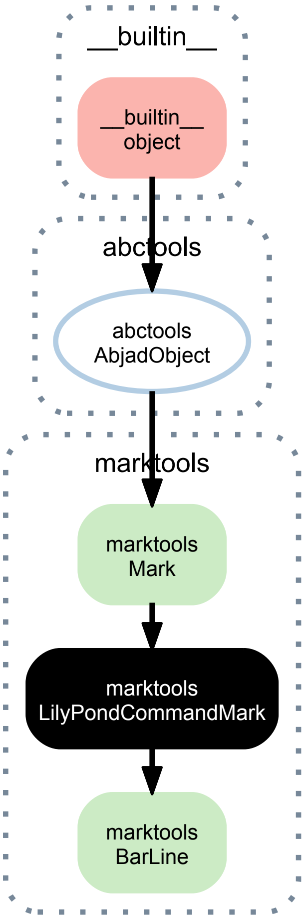
Raise exception.

BendAfter.**__ne__**(*arg*)

BendAfter.**__repr__**()

BendAfter.**__str__**()

15.2.5 marktools.LilyPondCommandMark



class `marktools.LilyPondCommandMark`(**args*)

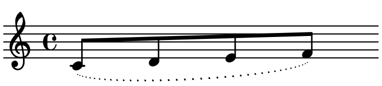
New in version 2.0. LilyPond command mark:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
```

```
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')(staff[0])
```

```
>>> f(staff)
\new Staff {
    \slurDotted
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(staff)
```



Initialize LilyPond command marks from command name; or from command name with format slot; or from another LilyPond command mark; or from another LilyPond command mark with format slot.

LilyPond command marks implement `__slots__`.

Read-only properties

`LilyPondCommandMark.`**`lilypond_format`**
Read-only LilyPond input format of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')(note)
>>> lilypond_command.lilypond_format
'\slurDotted'
```

Return string.

`LilyPondCommandMark.`**`start_component`**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`LilyPondCommandMark.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`LilyPondCommandMark.`**`command_name`**
Get command name of LilyPond command mark:

```
>>> lilypond_command = marktools.LilyPondCommandMark('slurDotted')
>>> lilypond_command.command_name
'slurDotted'
```

Set command name of LilyPond command mark:

```
>>> lilypond_command.command_name = 'slurDashed'
>>> lilypond_command.command_name
'slurDashed'
```

Set string.

`LilyPondCommandMark.`**`format_slot`**
New in version 2.3. Get format slot of LilyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('break', 'after')
>>> lilypond_command.format_slot
'after'
```

Set format slot of LiyPond command mark:

```
>>> note = Note("c'4")
>>> lilypond_command = marktools.LilyPondCommandMark('break', 'after')
>>> lilypond_command.format_slot = 'before'
>>> lilypond_command.format_slot
'before'
```

Set to `'before'`, `'after'`, `'opening'`, `'closing'`, `'right'` or none.

Methods

`LilyPondCommandMark.`**`attach`**`(`*start_component*`)`
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

`LilyPondCommandMark.`**`detach`**`()`

Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

`LilyPondCommandMark.`**`__call__`**`(`*`*args`*`)`

`LilyPondCommandMark.`**`__copy__`**`(`*`*args`*`)`

`LilyPondCommandMark.`**`__deepcopy__`**`(`*`*args`*`)`

`LilyPondCommandMark.`**`__delattr__`**`(`*`*args`*`)`

`LilyPondCommandMark.`**`__eq__`**`(`*`arg`*`)`

`LilyPondCommandMark.`**`__ge__`**`(`*`expr`*`)`

Abjad objects by default do not implement this method.

Raise exception.

`LilyPondCommandMark.`**`__gt__`**`(`*`expr`*`)`

Abjad objects by default do not implement this method.

Raise exception

`LilyPondCommandMark.`**`__le__`**`(`*`expr`*`)`

Abjad objects by default do not implement this method.

Raise exception.

`LilyPondCommandMark.`**`__lt__`**`(`*`expr`*`)`

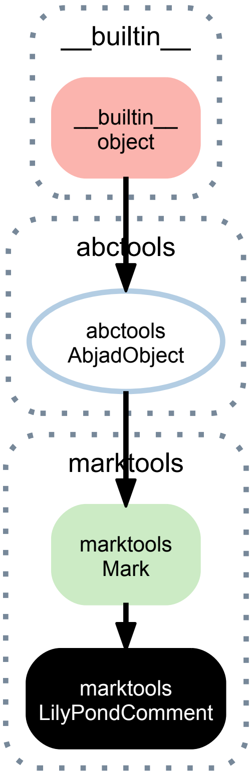
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondCommandMark.`**`__ne__`**`(`*`arg`*`)`

`LilyPondCommandMark.`**`__repr__`**`()`

15.2.6 marktools.LilyPondComment



class marktools.**LilyPondComment**(**args*)

New in version 2.0. User-defined LilyPond comment:

```
>>> note = Note("c'4")
```

```
>>> marktools.LilyPondComment('this is a comment')(note)
LilyPondComment('this is a comment')(c'4)
```

```
>>> f(note)
% this is a comment
c'4
```

Initialize LilyPond comment from contents string; or contents string and format slot; or from other LilyPond comment; or from other LilyPond comment and format slot.

LilyPond comments implement `__slots__`.

Read-only properties

LilyPondComment.**lilypond_format**

Read-only LilyPond input format of comment:

```
>>> comment = marktools.LilyPondComment('this is a comment.')
>>> comment.lilypond_format
'% this is a comment.'
```

Return string.

LilyPondComment.**start_component**

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

LilyPondComment.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

LilyPondComment.**contents_string**
Get contents string of comment:

```
>>> comment = marktools.LilyPondComment('comment contents string')
>>> comment.contents_string
'comment contents string'
```

Set contents string of comment:

```
>>> comment.contents_string = 'new comment contents string'
>>> comment.contents_string
'new comment contents string'
```

Set string.

LilyPondComment.**format_slot**
New in version 2.3. Get format slot of LilyPond comment:

```
>>> note = Note("c'4")
>>> lilypond_comment = marktools.LilyPondComment('comment')
>>> lilypond_comment.format_slot
'before'
```

Set format slot of LiyPond comment:

```
>>> note = Note("c'4")
>>> lilypond_comment = marktools.LilyPondComment('comment')
>>> lilypond_comment.format_slot = 'after'
>>> lilypond_comment.format_slot
'after'
```

Set to `'before'`, `'after'`, `'opening'`, `'closing'`, `'right'` or none.

Methods

LilyPondComment.**attach**(*start_component*)
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

LilyPondComment.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

LilyPondComment.**__call__**(*args*)

LilyPondComment.**__copy__**(*args*)

LilyPondComment.**__deepcopy__**(*args*)

LilyPondComment.**__delattr__**(*args*)

LilyPondComment.**__eq__**(*arg*)

LilyPondComment.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LilyPondComment.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

LilyPondComment.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LilyPondComment.**__lt__**(*expr*)

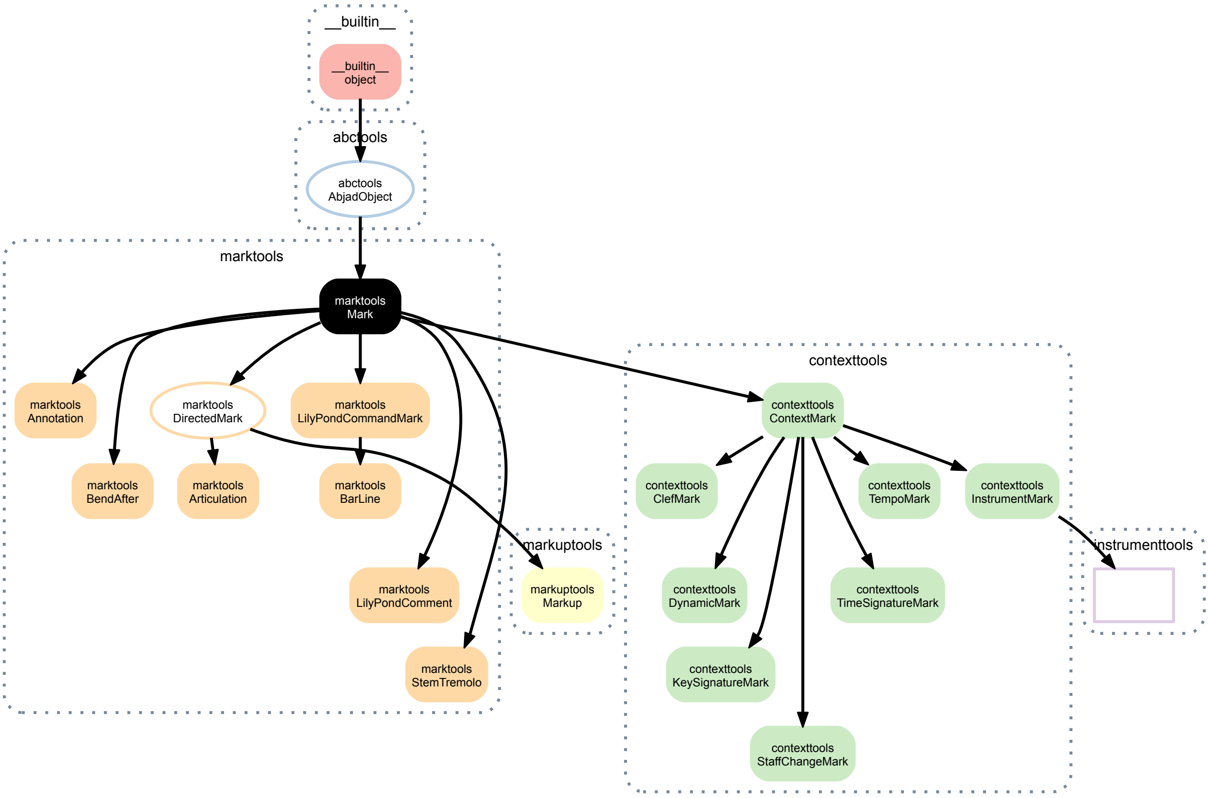
Abjad objects by default do not implement this method.

Raise exception.

LilyPondComment.**__ne__**(*arg*)

LilyPondComment.**__repr__**()

15.2.7 marktools.Mark



class `marktools.Mark(*args)`

New in version 2.0. Abstract class from which concrete marks inherit:

```
>>> note = Note("c'4")
```

```
>>> marktools.Mark()(note)
Mark()(c'4)
```

Marks override `__call__` to attach to a note, rest or chord.

Marks are immutable.

Read-only properties

`Mark.start_component`

Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

`Mark.storage_format`

Storage format of Abjad object.

Return string.

Methods

`Mark.attach(start_component)`

Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

`Mark.detach()`

Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

`Mark.__call__(*args)`

`Mark.__copy__(*args)`

`Mark.__deepcopy__(*args)`

`Mark.__delattr__(*args)`

`Mark.__eq__(arg)`

`Mark.__ge__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`Mark.__gt__(expr)`

Abjad objects by default do not implement this method.

Raise exception

`Mark.__le__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`Mark.__lt__(expr)`

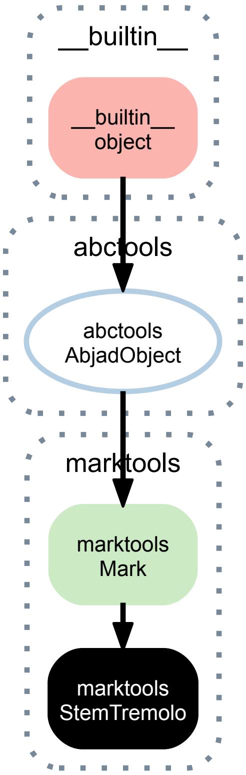
Abjad objects by default do not implement this method.

Raise exception.

`Mark.__ne__(arg)`

`Mark.__repr__()`

15.2.8 marktools.StemTremolo



class `marktools.StemTremolo(*args)`
New in version 2.0. Abjad model of stem tremolo:

```
>>> note = Note("c'4")
```

```
>>> marktools.StemTremolo(16)(note)
StemTremolo(16)(c'4)
```

```
>>> f(note)
c'4 :16
```

```
>>> show(note)
```

Stem tremolos implement `__slots__`.

Read-only properties

`StemTremolo.lilypond_format`
Read-only LilyPond format string:

```
>>> stem_tremolo = marktools.StemTremolo(16)
>>> stem_tremolo.lilypond_format
':16'
```

Return string.

`StemTremolo.start_component`
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

StemTremolo.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

StemTremolo.**tremolo_flags**
Get tremolo flags:

```
>>> stem_tremolo = marktools.StemTremolo(16)
>>> stem_tremolo.tremolo_flags
16
```

Set tremolo flags:

```
>>> stem_tremolo.tremolo_flags = 32
>>> stem_tremolo.tremolo_flags
32
```

Set integer.

Methods

StemTremolo.**attach**(*start_component*)
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

StemTremolo.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

StemTremolo.**__call__**(**args*)

StemTremolo.**__copy__**(**args*)
Copy stem tremolo:

```
>>> import copy
>>> stem_tremolo_1 = marktools.StemTremolo(16)
>>> stem_tremolo_2 = copy.copy(stem_tremolo_1)
```

```
>>> stem_tremolo_1 == stem_tremolo_2
True
```

```
>>> stem_tremolo_1 is not stem_tremolo_2
True
```

Return new stem tremolo.

StemTremolo.**__deepcopy__**(**args*)
Copy stem tremolo:

```
>>> import copy
>>> stem_tremolo_1 = marktools.StemTremolo(16)
>>> stem_tremolo_2 = copy.copy(stem_tremolo_1)
```

```
>>> stem_tremolo_1 == stem_tremolo_2
True
```

```
>>> stem_tremolo_1 is not stem_tremolo_2
True
```

Return new stem tremolo.

StemTremolo.**__delattr__**(**args*)

StemTremolo.**__eq__**(*expr*)
True when *expr* is a stem tremolo with equal tremolo flags: Otherwise false:

```
>>> stem_tremolo_1 = marktools.StemTremolo(16)
>>> stem_tremolo_2 = marktools.StemTremolo(16)
>>> stem_tremolo_3 = marktools.StemTremolo(32)
```

```
>>> stem_tremolo_1 == stem_tremolo_1
True
>>> stem_tremolo_1 == stem_tremolo_2
True
>>> stem_tremolo_1 == stem_tremolo_3
False
>>> stem_tremolo_2 == stem_tremolo_1
True
>>> stem_tremolo_2 == stem_tremolo_2
True
>>> stem_tremolo_2 == stem_tremolo_3
False
>>> stem_tremolo_3 == stem_tremolo_1
False
>>> stem_tremolo_3 == stem_tremolo_2
False
>>> stem_tremolo_3 == stem_tremolo_3
True
```

Return boolean.

StemTremolo.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StemTremolo.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`StemTremolo.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`StemTremolo.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`StemTremolo.__ne__(arg)`

`StemTremolo.__repr__()`

`StemTremolo.__str__()`

15.3 Functions

15.3.1 marktools.attach_annotations_to_components_in_expr

`marktools.attach_annotations_to_components_in_expr(expr, annotations)`
New in version 2.3. Attach *annotations* to components in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> annotation = marktools.Annotation('foo', 'bar')
>>> marktools.attach_annotations_to_components_in_expr(staff.leaves, [annotation])
```

```
>>> for x in staff:
...     print x, marktools.get_annotations_attached_to_component(x)
...
c'8 (Annotation('foo', 'bar')(c'8),)
d'8 (Annotation('foo', 'bar')(d'8),)
e'8 (Annotation('foo', 'bar')(e'8),)
f'8 (Annotation('foo', 'bar')(f'8),)
```

Return none.

15.3.2 marktools.attach_articulations_to_notes_and_chords_in_expr

`marktools.attach_articulations_to_notes_and_chords_in_expr(expr, articulations)`
New in version 2.0. Attach *articulations* to notes and chords in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.attach_articulations_to_notes_and_chords_in_expr(staff, list('^.'))
```

```
>>> f(staff)
\new Staff {
    c'8 -\marcato -\staccato
    d'8 -\marcato -\staccato
    e'8 -\marcato -\staccato
    f'8 -\marcato -\staccato
}
```

Return none.

15.3.3 marktools.attach_lilypond_command_marks_to_components_in_expr

`marktools.attach_lilypond_command_marks_to_components_in_expr(expr, lilypond_command_marks)`
New in version 2.3. Attach *lilypond_command_marks* to components in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_command_mark = marktools.LilyPondCommandMark('stemUp')
>>> marktools.attach_lilypond_command_marks_to_components_in_expr(
...     staff.leaves, [lilypond_command_mark])
```

```
>>> f(staff)
\new Staff {
    \stemUp
    c'8
    \stemUp
    d'8
    \stemUp
    e'8
    \stemUp
    f'8
}
```

Return none.

15.3.4 marktools.attach_lilypond_comments_to_components_in_expr

`marktools.`**`attach_lilypond_comments_to_components_in_expr`**(*expr*, *lilypond_comments*)

New in version 2.3. Attach *lilypond_comments* to components in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_comment = marktools.LilyPondComment('foo', 'right')
>>> marktools.attach_lilypond_comments_to_components_in_expr(
...     staff.leaves, [lilypond_comment])
```

```
>>> f(staff)
\new Staff {
    c'8 % foo
    d'8 % foo
    e'8 % foo
    f'8 % foo
}
```

Return none.

15.3.5 marktools.attach_stem_tremolos_to_notes_and_chords_in_expr

`marktools.`**`attach_stem_tremolos_to_notes_and_chords_in_expr`**(*expr*, *stem_tremolos*)

New in version 2.3. Attach *stem_tremolos* to notes and chords in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> stem_tremolo = marktools.StemTremolo(16)
>>> marktools.attach_stem_tremolos_to_notes_and_chords_in_expr(staff, [stem_tremolo])
```

```
>>> f(staff)
\new Staff {
    c'8 :16
    d'8 :16
    e'8 :16
    f'8 :16
}
```

Return none.

15.3.6 marktools.detach_annotations_attached_to_component

`marktools.`**`detach_annotations_attached_to_component`**(*component*)

New in version 2.0. Detach annotations attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.Annotation('annotation 1')(staff[0])
Annotation('annotation 1')(c'8)
>>> marktools.Annotation('annotation 2')(staff[0])
Annotation('annotation 2')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.get_annotations_attached_to_component(staff[0])
(Annotation('annotation 1')(c'8), Annotation('annotation 2')(c'8))
```

```
>>> marktools.detach_annotations_attached_to_component(staff[0])
(Annotation('annotation 1'), Annotation('annotation 2'))
```

```
>>> marktools.get_annotations_attached_to_component(staff[0])
()
```

Return tuple or zero or more annotations detached.

15.3.7 marktools.detach_articulations_attached_to_component

marktools.**detach_articulations_attached_to_component**(*component*)

New in version 2.0. Detach articulations attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.Articulation('^')(staff[0])
Articulation('^')(c'8)
>>> marktools.Articulation('.')(staff[0])
Articulation('.')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 -\marcato -\staccato (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.get_articulations_attached_to_component(staff[0])
(Articulation('^')(c'8), Articulation('.')(c'8))
```

```
>>> marktools.detach_articulations_attached_to_component(staff[0])
(Articulation('^'), Articulation('.'))
```

```
>>> marktools.get_articulations_attached_to_component(staff[0])
()
```

Return tuple or zero or more articulations detached.

15.3.8 marktools.detach_lilypond_command_marks_attached_to_component

marktools.**detach_lilypond_command_marks_attached_to_component**(*component*, *command_name=None*)

New in version 2.0. Detach LilyPond command marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.LilyPondCommandMark('slurDotted')(staff[0])
LilyPondCommandMark('slurDotted')(c'8)
>>> marktools.LilyPondCommandMark('slurUp')(staff[0])
LilyPondCommandMark('slurUp')(c'8)
```

```
>>> f(staff)
\new Staff {
    \slurDotted
    \slurUp
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.detach_lilypond_command_marks_attached_to_component(staff[0])
(LilyPondCommandMark('slurDotted'), LilyPondCommandMark('slurUp'))
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

Return tuple of zero or more marks detached.

15.3.9 marktools.detach_lilypond_comments_attached_to_component

marktools.**detach_lilypond_comments_attached_to_component**(*component*)

New in version 2.0. Detach LilyPond comments attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.LilyPondComment('comment 1')(staff[0])
LilyPondComment('comment 1')(c'8)
>>> marktools.LilyPondComment('comment 2')(staff[0])
LilyPondComment('comment 2')(c'8)
```

```
>>> f(staff)
\new Staff {
    % comment 1
    % comment 2
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.detach_lilypond_comments_attached_to_component(staff[0])
(LilyPondComment('comment 1'), LilyPondComment('comment 2'))
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.get_lilypond_comments_attached_to_component(staff[0])
()
```

Return tuple or zero or more LilyPond comments.

15.3.10 marktools.detach_marks_attached_to_component

marktools.**detach_marks_attached_to_component**(*component*)

New in version 2.0. Detach marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.Articulation('^')(staff[0])
Articulation('^')(c'8)
>>> marktools.LilyPondComment('comment 1')(staff[0])
LilyPondComment('comment 1')(c'8)
>>> marktools.LilyPondCommandMark('slurUp')(staff[0])
LilyPondCommandMark('slurUp')(c'8)
```

```
>>> f(staff)
\new Staff {
    % comment 1
    \slurUp
    c'8 -\marcato (
    d'8
    e'8
    f'8 )
}
```

```
>>> for mark in marktools.get_marks_attached_to_component(staff[0]):
...     mark
...
Articulation('^')(c'8)
LilyPondComment('comment 1')(c'8)
LilyPondCommandMark('slurUp')(c'8)
```

```
>>> for mark in marktools.detach_marks_attached_to_component(staff[0]):
...     mark
...
Articulation('^')
LilyPondComment('comment 1')
LilyPondCommandMark('slurUp')
```

```
>>> marktools.get_marks_attached_to_component(staff[0])
()
```

Return tuple or zero or more marks detached.

15.3.11 marktools.detach_marks_attached_to_components_in_expr

marktools.**detach_marks_attached_to_components_in_expr**(*expr*)

New in version 2.9. Detach marks attached to components in *expr*:

```
>>> staff = Staff("c'4 \staccato d' \marcato e' \staccato f' \marcato")
```

```
>>> f(staff)
\new Staff {
    c'4 -\staccato
    d'4 -\marcato
    e'4 -\staccato
    f'4 -\marcato
}
```

```
>>> for mark in marktools.detach_marks_attached_to_components_in_expr(staff):
...     mark
...
Articulation('staccato')
Articulation('marcato')
Articulation('staccato')
Articulation('marcato')
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more detached marks.

15.3.12 marktools.detach_noncontext_marks_attached_to_component

marktools.**detach_noncontext_marks_attached_to_component**(*component*)
New in version 2.3. Detach noncontext marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((2, 4))(staff[0])
TimeSignatureMark((2, 4))(c'8)
>>> marktools.Articulation('staccato')(staff[0])
Articulation('staccato')(c'8)
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'8 -\staccato
    d'8
    e'8
    f'8
}
```

```
>>> marktools.detach_noncontext_marks_attached_to_component(staff[0])
(Articulation('staccato'),)
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'8
    d'8
    e'8
    f'8
}
```

Return tuple of noncontext marks.

15.3.13 marktools.detach_stem_tremolos_attached_to_component

marktools.**detach_stem_tremolos_attached_to_component**(*component*)
New in version 2.0. Detach stem tremolos attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.StemTremolo(16)(staff[0])
StemTremolo(16)(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 :16
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_stem_tremolos_attached_to_component(staff[0])
(StemTremolo(16)(c'8),)
```

```
>>> marktools.detach_stem_tremolos_attached_to_component(staff[0])
(StemTremolo(16),)
```

```
>>> marktools.get_stem_tremolos_attached_to_component(staff[0])
()
```

Return tuple or zero or more stem tremolos detached.

15.3.14 marktools.get_annotation_attached_to_component

marktools.**get_annotation_attached_to_component**(*component*)
New in version 2.0. Get exactly one annotation attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Annotation('special information')(staff[0])
Annotation('special information')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_annotation_attached_to_component(staff[0])
Annotation('special information')(c'8)
```

Return one annotation.

Raise missing mark error when no annotation is attached.

Raise extra mark error when more than one annotation is attached.

15.3.15 marktools.get_annotations_attached_to_component

marktools.**get_annotations_attached_to_component**(*component*)
New in version 2.0. Get annotations attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Annotation('annotation 1')(staff[0])
Annotation('annotation 1')(c'8)
>>> marktools.Annotation('annotation 2')(staff[0])
Annotation('annotation 2')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_annotations_attached_to_component(staff[0])
(Annotation('annotation 1')(c'8), Annotation('annotation 2')(c'8))
```

Return tuple of zero or more annotations.

15.3.16 marktools.get_articulation_attached_to_component

marktools.**get_articulation_attached_to_component**(*component*)
New in version 2.0. Get exactly one articulation attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Articulation('staccato')(staff[0])
Articulation('staccato')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 -\staccato
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_articulation_attached_to_component(staff[0])
Articulation('staccato')(c'8)
```

Return one articulation.

Raise missing mark error when no articulation is attached.

Raise extra mark error when more than one articulation is attached.

15.3.17 marktools.get_articulations_attached_to_component

marktools.**get_articulations_attached_to_component**(*component*)

New in version 2.0. Get articulations attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Articulation('staccato')(staff[0])
Articulation('staccato')(c'8)
>>> marktools.Articulation('marcato')(staff[0])
Articulation('marcato')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 -\marcato -\staccato
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_articulations_attached_to_component(staff[0])
(Articulation('staccato')(c'8), Articulation('marcato')(c'8))
```

Return tuple of zero or more articulations.

15.3.18 marktools.get_lilypond_command_mark_attached_to_component

marktools.**get_lilypond_command_mark_attached_to_component**(*component*)

New in version 2.0. Get exactly one LilyPond command mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.LilyPondCommandMark('stemUp')(staff[0])
LilyPondCommandMark('stemUp')(c'8)
```

```
>>> f(staff)
\new Staff {
    \stemUp
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_lilypond_command_mark_attached_to_component(staff[0])
LilyPondCommandMark('stemUp')(c'8)
```

Return one LilyPond command mark.

Raise missing mark error when no LilyPond command mark is attached.

Raise extra mark error when more than one LilyPond command mark is attached.

15.3.19 marktools.get_lilypond_command_marks_attached_to_component

marktools.**get_lilypond_command_marks_attached_to_component**(*component*, *command_name=None*)

New in version 2.0. Get LilyPond command marks attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.LilyPondCommandMark('slurDotted')(staff[0])
LilyPondCommandMark('slurDotted')(c'8)
>>> marktools.LilyPondCommandMark('slurUp')(staff[0])
LilyPondCommandMark('slurUp')(c'8)
```

```
>>> f(staff)
\new Staff {
    \slurDotted
    \slurUp
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.get_lilypond_command_marks_attached_to_component(staff[0])
(LilyPondCommandMark('slurDotted')(c'8), LilyPondCommandMark('slurUp')(c'8))
```

Return tuple of zero or more marks.

15.3.20 marktools.get_lilypond_comment_attached_to_component

marktools.**get_lilypond_comment_attached_to_component**(*component*)

New in version 2.0. Get exactly one LilyPond comment attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.LilyPondComment('comment')(staff[0])
LilyPondComment('comment')(c'8)
```

```
>>> f(staff)
\new Staff {
    % comment
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_lilypond_comment_attached_to_component(staff[0])
LilyPondComment('comment')(c'8)
```

Return one LilyPond comment.

Raise missing mark error when no LilyPond comment is attached.

Raise extra mark error when more than one LilyPond comment is attached.

15.3.21 marktools.get_lilypond_comments_attached_to_component

marktools.**get_lilypond_comments_attached_to_component**(*component*)

New in version 2.0. Get LilyPond comments attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> marktools.LilyPondComment('comment 1')(staff[0])
LilyPondComment('comment 1')(c'8)
>>> marktools.LilyPondComment('comment 2')(staff[0])
LilyPondComment('comment 2')(c'8)
```

```
>>> f(staff)
\new Staff {
    % comment 1
    % comment 2
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> marktools.get_lilypond_comments_attached_to_component(staff[0])
(LilyPondComment('comment 1')(c'8), LilyPondComment('comment 2')(c'8))
```

Return tuple of zero or more LilyPond comments.

15.3.22 marktools.get_mark_attached_to_component

marktools.**get_mark_attached_to_component**(*component*)
New in version 2.0. Get exactly one mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Mark()(staff[0])
Mark()(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_mark_attached_to_component(staff[0])
Mark()(c'8)
```

Return one mark.

Raise missing mark error when no mark is attached.

Raise extra mark error when more than one mark is attached.

15.3.23 marktools.get_marks_attached_to_component

marktools.**get_marks_attached_to_component**(*component*)
New in version 2.0. Get all marks attached to *component*‘:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> comment_mark = marktools.LilyPondComment('beginning of note content')(staff[0])
>>> marktools.LilyPondCommandMark('slurDotted')(staff[0])
LilyPondCommandMark('slurDotted')(c'8)
```

```
>>> f(staff)
\new Staff {
    % beginning of note content
    \slurDotted
    c'8 (
    d'8
    e'8
```

```
    f'8 )
}
```

```
>>> marktools.get_marks_attached_to_component(staff[0])
(LilyPondComment('beginning of note content')(c'8), LilyPondCommandMark('slurDotted')(c'8))
```

Return tuple of zero or more marks.

15.3.24 marktools.get_marks_attached_to_components_in_expr

marktools.**get_marks_attached_to_components_in_expr**(*expr*)
New in version 2.9. Get marks attached to components in *expr*:

```
>>> staff = Staff(r"c'4 \pp d' \staccato e' \ff f' \staccato")
```

```
\new Staff {
    c'4 \pp
    d'4 -\staccato
    e'4 \ff
    f'4 -\staccato
}
```

```
>>> for mark in marktools.get_marks_attached_to_components_in_expr(staff):
...     mark
...
DynamicMark('pp')(c'4)
Articulation('staccato')(d'4)
DynamicMark('ff')(e'4)
Articulation('staccato')(f'4)
```

Return tuple of zero or more marks.

15.3.25 marktools.get_noncontext_mark_attached_to_component

marktools.**get_noncontext_mark_attached_to_component**(*component*)
New in version 2.0. Get exactly one noncontext_mark attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Articulation('staccato')(staff[0])
Articulation('staccato')(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 -\staccato
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_noncontext_mark_attached_to_component(staff[0])
Articulation('staccato')(c'8)
```

Return one noncontext_mark.

Raise missing mark error when no noncontext_mark is attached.

Raise extra mark error when more than one noncontext_mark is attached.

15.3.26 marktools.get_noncontext_marks_attached_to_component

marktools.**get_noncontext_marks_attached_to_component**(*component*)
New in version 2.0. Get noncontext marks attached to component:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> contexttools.TimeSignatureMark((2, 4))(staff[0])
TimeSignatureMark((2, 4))(c'8)
>>> marktools.Articulation('staccato')(staff[0])
Articulation('staccato')(c'8)
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'8 -\staccato
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_noncontext_marks_attached_to_component(staff[0])
(Articulation('staccato')(c'8),)
```

Return tuple of zero or more marks.

15.3.27 marktools.get_stem_tremolo_attached_to_component

marktools.**get_stem_tremolo_attached_to_component**(*component*)

New in version 2.0. Get stem tremolo attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.StemTremolo(16)(staff[0])
StemTremolo(16)(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 :16
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_stem_tremolo_attached_to_component(staff[0])
StemTremolo(16)(c'8)
```

Raise missing mark error when no stem tremolo attaches to *component*.

Raise extra mark error when more than one stem tremolo attaches to *component*.

Return stem tremolo.

15.3.28 marktools.get_stem_tremolos_attached_to_component

marktools.**get_stem_tremolos_attached_to_component**(*component*, *tremolo_flags=None*)

New in version 2.3. Get stem tremolos attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.StemTremolo(16)(staff[0])
StemTremolo(16)(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 :16
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_stem_tremolos_attached_to_component(staff[0])
(StemTremolo(16)(c'8),)
```

Return tuple of zero or more stem tremolos.

15.3.29 marktools.get_value_of_annotation_attached_to_component

marktools.**get_value_of_annotation_attached_to_component**(*component*, *name*, *default_value=None*)

New in version 2.0. Get value of annotation with *name* attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> marktools.Annotation('special dictionary', {})(staff[0])
Annotation('special dictionary', {})(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> marktools.get_value_of_annotation_attached_to_component(staff[0], 'special dictionary')
{}
```

Return arbitrary value of annotation.

Return *default_value* when no annotation with *name* is attached.

Raise extra mark error when more than one annotation with *name* is attached.

15.3.30 marktools.is_component_with_annotation_attached

marktools.**is_component_with_annotation_attached**(*expr*, *annotation_name=None*, *annotation_value=None*)

New in version 2.3. True when *expr* is component with annotation attached:

```
>>> note = Note("c'4")
>>> marktools.Annotation('foo', 'bar')(note)
Annotation('foo', 'bar')(c'4)
```

```
>>> marktools.is_component_with_annotation_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_annotation_attached(note)
False
```

Return boolean.

15.3.31 marktools.is_component_with_articulation_attached

marktools.**is_component_with_articulation_attached**(*expr*, *articulation_name=None*)

New in version 2.3. True when *expr* is component with articulation attached:

```
>>> note = Note("c'4")
>>> marktools.Articulation('staccato')(note)
Articulation('staccato')(c'4)
```

```
>>> marktools.is_component_with_articulation_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_articulation_attached(note)
False
```

Return boolean.

15.3.32 marktools.is_component_with_lilypond_command_mark_attached

marktools.**is_component_with_lilypond_command_mark_attached**(*expr*, *command_name=None*)

New in version 2.0. True when *expr* is component with LilyPond command mark attached:

```
>>> note = Note("c'4")
>>> marktools.LilyPondCommandMark('stemUp')(note)
LilyPondCommandMark('stemUp')(c'4)
```

```
>>> marktools.is_component_with_lilypond_command_mark_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_lilypond_command_mark_attached(note)
False
```

Return boolean.

15.3.33 marktools.is_component_with_lilypond_comment_attached

marktools.**is_component_with_lilypond_comment_attached**(*expr*, *comment_contents_string=None*)

New in version 2.3. True when *expr* is component with LilyPond comment mark attached:

```
>>> note = Note("c'4")
>>> marktools.LilyPondComment('comment here')(note)
LilyPondComment('comment here')(c'4)
```

```
>>> marktools.is_component_with_lilypond_comment_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_lilypond_comment_attached(note)
False
```

Return boolean.

15.3.34 marktools.is_component_with_mark_attached

marktools.**is_component_with_mark_attached**(*expr*)

New in version 2.3. True when *expr* is component with mark attached:

```
>>> note = Note("c'4")
>>> marktools.Mark()(note)
Mark()(c'4)
```

```
>>> marktools.is_component_with_mark_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_mark_attached(note)
False
```

Return boolean.

15.3.35 marktools.is_component_with_noncontext_mark_attached

marktools.**is_component_with_noncontext_mark_attached**(*expr*)

New in version 2.3. True when *expr* is component with noncontext mark attached:

```
>>> note = Note("c'4")
>>> marktools.Articulation('staccato')(note)
Articulation('staccato')(c'4)
```

```
>>> marktools.is_component_with_noncontext_mark_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_noncontext_mark_attached(note)
False
```

Return boolean.

15.3.36 marktools.is_component_with_stem_tremolo_attached

marktools.**is_component_with_stem_tremolo_attached**(*expr*)

New in version 2.3. True when *expr* is component with LilyPond command mark attached:

```
>>> note = Note("c'4")
>>> marktools.StemTremolo(16)(note)
StemTremolo(16)(c'4)
```

```
>>> marktools.is_component_with_stem_tremolo_attached(note)
True
```

False otherwise:

```
>>> note = Note("c'4")
```

```
>>> marktools.is_component_with_stem_tremolo_attached(note)
False
```

Return boolean.

15.3.37 marktools.move_marks

marktools.**move_marks**(*donor*, *recipient*)

Move marks from *donor* component to *recipient* component:

```
>>> staff = Staff(r'\clef "bass" c \staccato d e f')
```

```
>>> f(staff)
\new Staff {
    \clef "bass"
    c4 -\staccato
    d4
    e4
    f4
}
```

```
>>> marktools.move_marks(staff[0], staff[2])
[Articulation('staccato')(e4), ClefMark('bass')(e4)]
```

```
>>> f(staff)
\new Staff {
    c4
    d4
    \clef "bass"
    e4 -\staccato
    f4
}
```

Return list of marks moved.

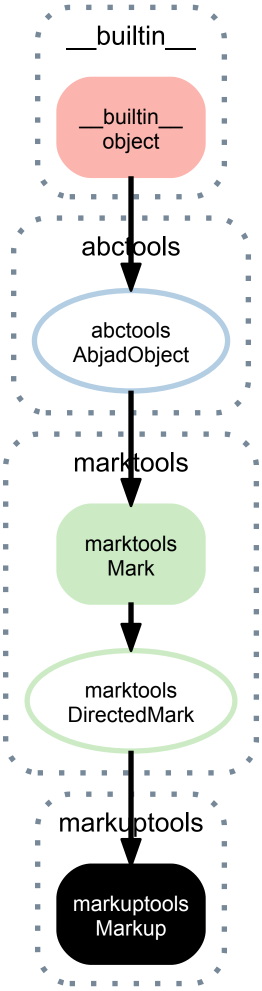
CHAPTER

SIXTEEN

MARKUPTOOLS

16.1 Concrete Classes

16.1.1 markuptools.Markup



class `markuptools.`**`Markup`**(*argument*, *direction=None*, *markup_name=None*)

Abjad model of LilyPond markup.

Initialize from string:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
```

```
>>> markup
Markup((MarkupCommand('bold', ['This is markup text.']),))
```

```
>>> f(markup)
\markup { \bold { "This is markup text." } }
```

```
>>> show(markup)
```

Initialize any markup from existing markup:

```
>>> markup_1 = markuptools.Markup('foo', direction=Up)
>>> markup_2 = markuptools.Markup(markup_1, direction=Down)
```

```
>>> f(markup_1)
^ \markup { foo }
```

```
>>> f(markup_2)
_ \markup { foo }
```

Attach markup to score components by calling them on the component:

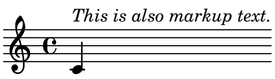
```
>>> note = Note("c'4")
```

```
>>> markup = markuptools.Markup(r'\italic { "This is also markup text." }', direction=Up)
```

```
>>> markup(note)
Markup((MarkupCommand('italic', ['This is also markup text.']),), direction=Up)(c'4)
```

```
>>> f(note)
c'4 ^ \markup { \italic { "This is also markup text." } }
```

```
>>> show(note)
```



Set *direction* to `Up`, `Down`, `'neutral'`, `'^'`, `'_'`, `'-'` or None.

Markup objects are immutable.

Return markup instance.

Read-only properties

`Markup.`**`contents`**

Read-only tuple of contents of markup:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
>>> markup.contents
(MarkupCommand('bold', ['This is markup text.']),)
```

Return string

`Markup.`**`indented_lilypond_format`**

Read-only indented LilyPond format of markup:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
>>> print markup.indented_lilypond_format
\markup {
    \bold {
        "This is markup text."
        }
    }
```

Return string.

Markup.**lilypond_format**
Read-only LilyPond format of markup:

```
>>> markup = markuptools.Markup(r'\bold { "This is markup text." }')
>>> markup.lilypond_format
'\\markup { \\bold { "This is markup text." } }'
```

Return string.

Markup.**markup_name**
Read-only name of markup:

```
>>> markup = markuptools.Markup(
...     r'\bold { allegro ma non troppo }', markup_name='non troppo')
```

```
>>> markup.markup_name
'non troppo'
```

Return string or none.

Markup.**start_component**
Read-only reference to mark start component:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

Return component or none.

Markup.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

Markup.**direction**

Methods

Markup.**attach**(*start_component*)
Attach mark to *start_component*:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()
```

```
>>> mark.attach(note)
Mark()(c'4)
```

```
>>> mark.start_component
Note("c'4")
```

Return mark.

Markup.**detach**()
Detach mark:

```
>>> note = Note("c'4")
>>> mark = marktools.Mark()(note)
```

```
>>> mark.start_component
Note("c'4")
```

```
>>> mark.detach()
Mark()
```

```
>>> mark.start_component is None
True
```

Return mark.

Special methods

Markup.**__call__**(**args*)

Markup.**__copy__**(**args*)

Markup.**__deepcopy__**(**args*)

Markup.**__delattr__**(**args*)

Markup.**__eq__**(*expr*)

Markup.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Markup.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Markup.**__hash__**()

Markup.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Markup.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

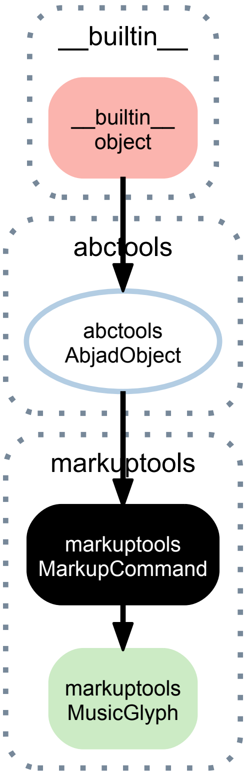
Raise exception.

Markup.**__ne__**(*expr*)

Markup.**__repr__**()

Markup.**__str__**()

16.1.2 markuptools.MarkupCommand



class markuptools.**MarkupCommand**(*command*, **args*)

Abjad model of a LilyPond markup command:

```
>>> circle = markuptools.MarkupCommand('draw-circle', 2.5, 0.1, False)
>>> square = markuptools.MarkupCommand('rounded-box', 'hello?')
>>> line = markuptools.MarkupCommand('line', [square, 'wow!'])
>>> rotate = markuptools.MarkupCommand('rotate', 60, line)
>>> combine = markuptools.MarkupCommand('combine', rotate, circle)
```

```
>>> print combine
\combine \rotate #60 \line { \rounded-box hello? wow! } \draw-circle #2.5 #0.1 ##f
```

Insert a markup command in markup in order to attach it to score components:

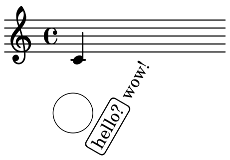
```
>>> note = Note("c'4")
```

```
>>> markup = markuptools.Markup(combine)
```

```
>>> markup = markup(note)
```

```
>>> f(note)
c'4
    - \markup {
        \combine
            \rotate
                #60
                \line
                    {
                        \rounded-box
                            hello?
                        wow!
                    }
            \draw-circle
                #2.5
                #0.1
                ##f
        }
```

```
>>> show(note)
```



Markup commands are immutable.

Return markup command.

Read-only properties

`MarkupCommand.`**`args`**
Read-only tuple of markup command arguments.

`MarkupCommand.`**`command`**
Read-only string of markup command command-name.

`MarkupCommand.`**`lilypond_format`**
Read-only format of markup command:

```
>>> markup_command = markuptools.MarkupCommand('draw-circle', 2.5, 0.1, False)
>>> markup_command.lilypond_format
'\\draw-circle #2.5 #0.1 ##f'
```

Returns string.

`MarkupCommand.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`MarkupCommand.`**`__eq__`**`(`*expr*`)`

`MarkupCommand.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MarkupCommand.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`MarkupCommand.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MarkupCommand.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

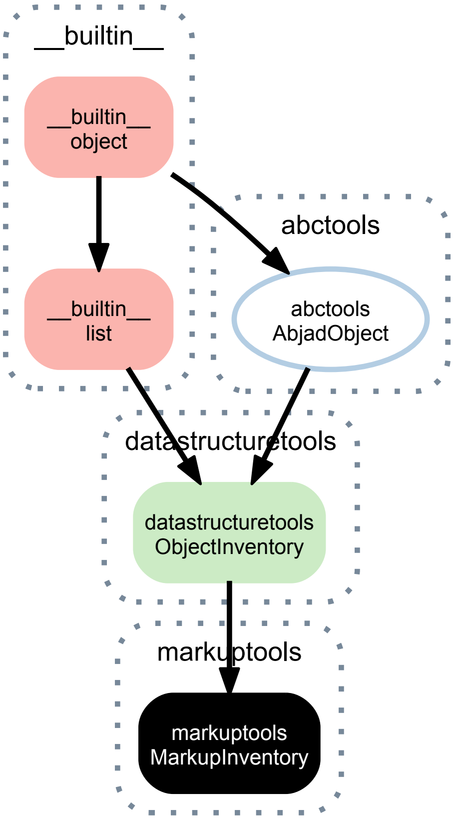
`MarkupCommand.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`MarkupCommand.`**`__repr__`**`()`

`MarkupCommand.`**`__str__`**`()`

16.1.3 markuptools.MarkupInventory



class `markuptools.`**`MarkupInventory`**(*tokens=None*, *name=None*)

New in version 2.8. Abjad model of an ordered list of markup:

```
>>> inventory = markuptools.MarkupInventory(['foo', 'bar'])
```

```
>>> inventory
MarkupInventory([Markup(('foo',)), Markup(('bar',))])
```

Markup inventories implement the list interface and are mutable.

Read-only properties

`MarkupInventory.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`MarkupInventory.`**`name`**

Read / write name of inventory.

Methods

`MarkupInventory.`**`append`**(*token*)

Change *token* to item and append.

`MarkupInventory.`**`count`**(*value*) → integer – return number of occurrences of value

MarkupInventory.**extend**(*tokens*)
Change *tokens* to items and extend.

MarkupInventory.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

MarkupInventory.**insert**()
L.insert(index, object) – insert object before index

MarkupInventory.**pop**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

MarkupInventory.**remove**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

MarkupInventory.**reverse**()
L.reverse() – reverse *IN PLACE*

MarkupInventory.**sort**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

MarkupInventory.**__add__**()
x.__add__(y) <==> x+y

MarkupInventory.**__contains__**(*token*)

MarkupInventory.**__delitem__**()
x.__delitem__(y) <==> del x[y]

MarkupInventory.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

MarkupInventory.**__eq__**()
x.__eq__(y) <==> x==y

MarkupInventory.**__ge__**()
x.__ge__(y) <==> x>=y

MarkupInventory.**__getitem__**()
x.__getitem__(y) <==> x[y]

MarkupInventory.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

MarkupInventory.**__gt__**()
x.__gt__(y) <==> x>y

MarkupInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

MarkupInventory.**__imul__**()
x.__imul__(y) <==> x*=y

MarkupInventory.**__iter__**() <==> *iter(x)*

MarkupInventory.**__le__**()
x.__le__(y) <==> x<=y

MarkupInventory.**__len__**() <==> *len(x)*

MarkupInventory.**__lt__**()
x.__lt__(y) <==> x<y

`MarkupInventory.__mul__()`
x.__mul__(n) <==> x*n

`MarkupInventory.__ne__()`
x.__ne__(y) <==> x!=y

`MarkupInventory.__repr__()`

`MarkupInventory.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

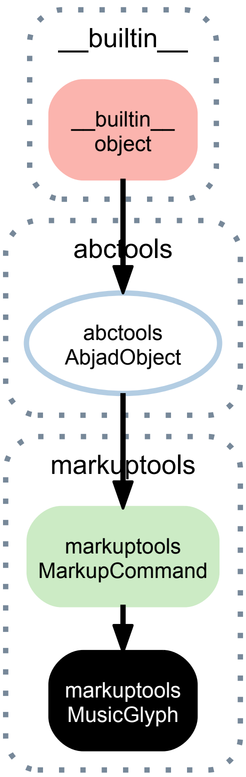
`MarkupInventory.__rmul__()`
x.__rmul__(n) <==> n*x

`MarkupInventory.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`MarkupInventory.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

16.1.4 markuptools.MusicGlyph



class `markuptools.MusicGlyph`(*glyph_name*)
Abjad model of a LilyPond musicglyph command:

```
>>> markuptools.MusicGlyph('accidentals.sharp')
MusicGlyph('accidentals.sharp')
>>> print _
\musicglyph #"accidentals.sharp"
```

Return *MusicGlyph* instance.

Read-only properties

MusicGlyph.**args**
Read-only tuple of markup command arguments.

MusicGlyph.**command**
Read-only string of markup command command-name.

MusicGlyph.**lilypond_format**
Read-only format of markup command:

```
>>> markup_command = markuptools.MarkupCommand('draw-circle', 2.5, 0.1, False)
>>> markup_command.lilypond_format
'\\draw-circle #2.5 #0.1 ##f'
```

Returns string.

MusicGlyph.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

MusicGlyph.**__eq__**(*expr*)

MusicGlyph.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MusicGlyph.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MusicGlyph.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MusicGlyph.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MusicGlyph.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MusicGlyph.**__repr__**()

MusicGlyph.**__str__**()

16.2 Functions

16.2.1 markuptools.all_are_markup

markuptools.**all_are_markup**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad markup:

```
>>> markup = markuptools.Markup('some text')
```

```
>>> markuptools.all_are_markup([markup])
True
```

True when *expr* is an empty sequence:

```
>>> markuptools.all_are_markup([])
True
```

Otherwise false:

```
>>> markuptools.all_are_markup('foo')
False
```

Return boolean.

16.2.2 markuptools.combine_markup_commands

markuptools.**combine_markup_commands**(**commands*)

Combine MarkupCommand and/or string objects.

LilyPond's 'combine' markup command can only take two arguments, so in order to combine more than two stencils, a cascade of 'combine' commands must be employed. *combine_markup_commands* simplifies this process.

```
>>> markup_a = markuptools.MarkupCommand('draw-circle', 4, 0.4, False)
>>> markup_b = markuptools.MarkupCommand(
...     'filled-box',
...     schemetools.SchemePair(-4, 4),
...     schemetools.SchemePair(-0.5, 0.5), 1)
>>> markup_c = "some text"
```

```
>>> markup = markuptools.combine_markup_commands(markup_a, markup_b, markup_c)
>>> result = markup.lilypond_format
```

```
>>> print result
\combine \combine \draw-circle #4 #0.4 ##f
    \filled-box #'(-4 . 4) #'(-0.5 . 0.5) #1 "some text"
```

Returns a markup command instance, or a string if that was the only argument.

16.2.3 markuptools.get_down_markup_attached_to_component

markuptools.**get_down_markup_attached_to_component**(*component*)

New in version 2.0. Get down-markup attached to component:

```
>>> chord = Chord([-11, 2, 5], (1, 4))
```

```
>>> markuptools.Markup('UP', Up)(chord)
Markup(('UP',), direction=Up)(<cs d' f'>4)
```

```
>>> markuptools.Markup('DOWN', Down)(chord)
Markup(('DOWN',), direction=Down)(<cs d' f'>4)
```

```
>>> markuptools.get_down_markup_attached_to_component(chord)
(Markup(('DOWN',), direction=Down)(<cs d' f'>4),)
```

Return tuple of zero or more markup objects.

16.2.4 markuptools.get_markup_attached_to_component

markuptools.**get_markup_attached_to_component**(*component*)

New in version 2.0. Get markup attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff[:])
```

```
>>> markuptools.Markup('foo')(staff[0])
Markup(('foo',))(c'8)
```

```
>>> markuptools.Markup('bar')(staff[0])
Markup(('bar',))(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 (
        - \markup {
            \column
                {
                    foo
                    bar
                }
            }
    d'8
    e'8
    f'8 )
}
```

```
>>> markuptools.get_markup_attached_to_component(staff[0])
(Markup(('foo',))(c'8), Markup(('bar',))(c'8))
```

Return tuple of zero or more markup objects.

16.2.5 markuptools.get_up_markup_attached_to_component

markuptools.**get_up_markup_attached_to_component**(*component*)
New in version 2.0. Get up-markup attached to component:

```
>>> chord = Chord([-11, 2, 5], (1, 4))
```

```
>>> markuptools.Markup('UP', Up)(chord)
Markup(('UP',), direction=Up)(<cs d' f'>4)
```

```
>>> markuptools.Markup('DOWN', Down)(chord)
Markup(('DOWN',), direction=Down)(<cs d' f'>4)
```

```
>>> markuptools.get_up_markup_attached_to_component(chord)
(Markup(('UP',), direction=Up)(<cs d' f'>4),)
```

Return tuple of zero or more markup objects.

16.2.6 markuptools.make_big_centered_page_number_markup

markuptools.**make_big_centered_page_number_markup**(*text=None*)
New in version 1.1. Make big centered page number markup:

```
>>> markup = markuptools.make_big_centered_page_number_markup()
```

```
>>> print markup.indented_lilypond_format
\markup {
    \fill-line
        {
            \bold
                \fontsize
                    #3
                    \concat
                        {
                            \on-the-fly
                                #print-page-number-check-first
```

```
                                    \fromproperty
                                        #'page:page-number-string
                                }
                }
            }
```

Return markup.

16.2.7 markuptools.make_blank_line_markup

markuptools.**make_blank_line_markup**()
New in version 2.9. Make blank line markup:

```
>>> markup = markuptools.make_blank_line_markup()
```

```
>>> markup
Markup((MarkupCommand('fill-line', [' ']),))
```

```
>>> f(markup)
\markup { \fill-line { " " } }
```

Return markup.

16.2.8 markuptools.make_centered_title_markup

markuptools.**make_centered_title_markup**(*title*, *font_name='Times'*, *font_size=18*, *vspace_before=6*, *vspace_after=12*)
New in version 2.9. Make centered *title* markup:

```
>>> markup = markuptools.make_centered_title_markup('String Quartet')
```

```
>>> print markup.indented_lilypond_format
\markup {
    \override
        #'(font-name . "Times")
        \fontsize
            #18
            \column
                {
                    \center-align
                        {
                            {
                                \vspace
                                    #6
                                \line
                                    {
                                        "String Quartet"
                                    }
                                \vspace
                                    #12
                            }
                        }
                }
    }
```

Return markup.

16.2.9 markuptools.make_vertically_adjusted_composer_markup

markuptools.**make_vertically_adjusted_composer_markup**(*composer*, *font_name='Times'*, *font_size=3*, *space_above=20*, *space_right=0*)

New in version 2.9. Make vertically adjusted *composer* markup:

```
>>> markup = markuptools.make_vertically_adjusted_composer_markup('Josquin Desprez')
```

```
>>> print markup.indented_lilypond_format
\markup {
    \override
        #'(font-name . "Times")
        {
            \hspace
                #0
            \raise
                #-20
                \fontsize
                    #3
                    "Josquin Desprez"
            \hspace
                #0
        }
    }
```

Return markup.

16.2.10 markuptools.remove_markup_attached_to_component

markuptools.**remove_markup_attached_to_component**(*component*)

New in version 2.0. Remove markup attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> slur = spannertools.SlurSpanner(staff[:])
>>> markuptools.Markup('foo')(staff[0])
Markup(('foo',))(c'8)
>>> markuptools.Markup('bar')(staff[0])
Markup(('bar',))(c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 (
        - \markup {
            \column
                {
                    foo
                    bar
                }
            }
    d'8
    e'8
    f'8 )
}
```

```
>>> markuptools.remove_markup_attached_to_component(staff[0])
(Markup(('foo',)), Markup(('bar',)))
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

Return tuple of zero or more markup objects.

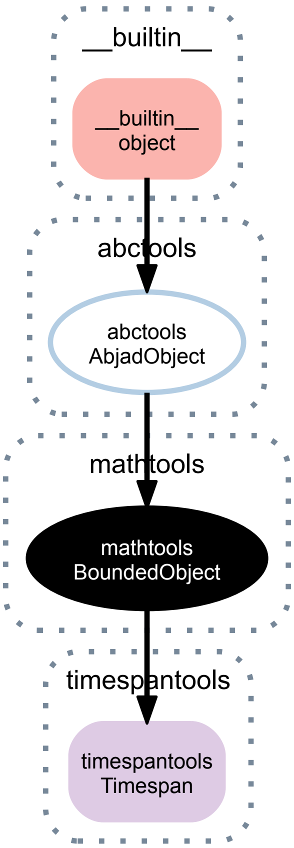
CHAPTER

SEVENTEEN

MATHTOOLS

17.1 Abstract Classes

17.1.1 mathtools.BoundedObject



class `mathtools.`**`BoundedObject`**

New in version 2.10. Bounded object mix-in.

Read-only properties

`BoundedObject.`**`is_closed`**

`BoundedObject.`**`is_half_closed`**

`BoundedObject.`**`is_half_open`**

`BoundedObject.`**`is_open`**

BoundedObject.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

BoundedObject.**is_left_closed**

BoundedObject.**is_left_open**

BoundedObject.**is_right_closed**

BoundedObject.**is_right_open**

Special methods

BoundedObject.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

BoundedObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BoundedObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BoundedObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BoundedObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BoundedObject.**__ne__**(*expr*)
Defined equal to the opposite of equality.

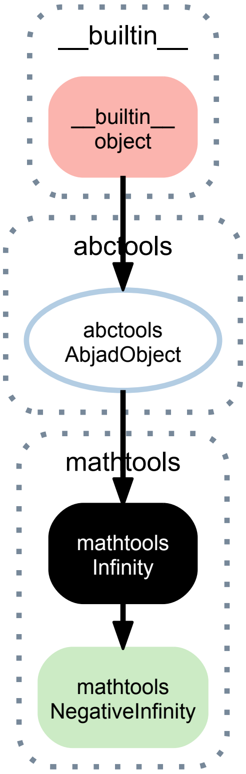
Return boolean.

BoundedObject.**__repr__**()
Interpreter representation of Abjad object.

Return string.

17.2 Concrete Classes

17.2.1 mathtools.Infinity



class `mathtools.`**`Infinity`**

Object-oriented infinity.

All numbers compare less than infinity:

```
>>> 9999999 < Infinity
True
```

```
>>> 2**38 < Infinity
True
```

Infinity compares equal to itself:

```
>>> Infinity == Infinity
True
```

Negative infinity compares less than infinity:

```
>>> NegativeInfinity < Infinity
True
```

Infinity is initialized at start-up and is available in the global Abjad namespace.

Read-only properties

`Infinity.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`Infinity.__eq__(expr)`

`Infinity.__ge__(expr)`

`Infinity.__gt__(expr)`

`Infinity.__le__(expr)`

`Infinity.__lt__(expr)`

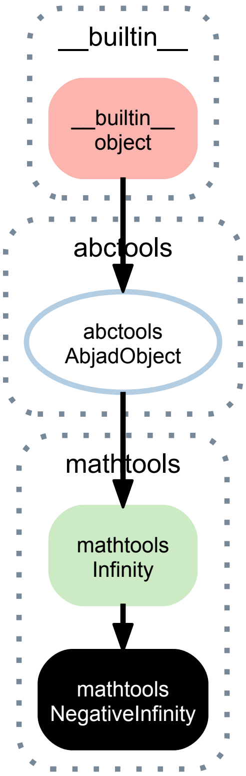
`Infinity.__ne__(expr)`

Defined equal to the opposite of equality.

Return boolean.

`Infinity.__repr__()`

17.2.2 mathtools.NegativeInfinity



class `mathtools.NegativeInfinity`

Object-oriented negative infinity.

All numbers compare greater than negative infinity:

```
>>> NegativeInfinity < -9999999
True
```

Negative infinity compares equal to itself:

```
>>> NegativeInfinity == NegativeInfinity
True
```

Negative infinity compares less than infinity:

```
>>> NegativeInfinity < Infinity
True
```

Negative infinity is initialize at start-up and is available in the global Abjad namespace.

Read-only properties

NegativeInfinity.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

NegativeInfinity.**__eq__**(*expr*)

NegativeInfinity.**__ge__**(*expr*)

NegativeInfinity.**__gt__**(*expr*)

NegativeInfinity.**__le__**(*expr*)

NegativeInfinity.**__lt__**(*expr*)

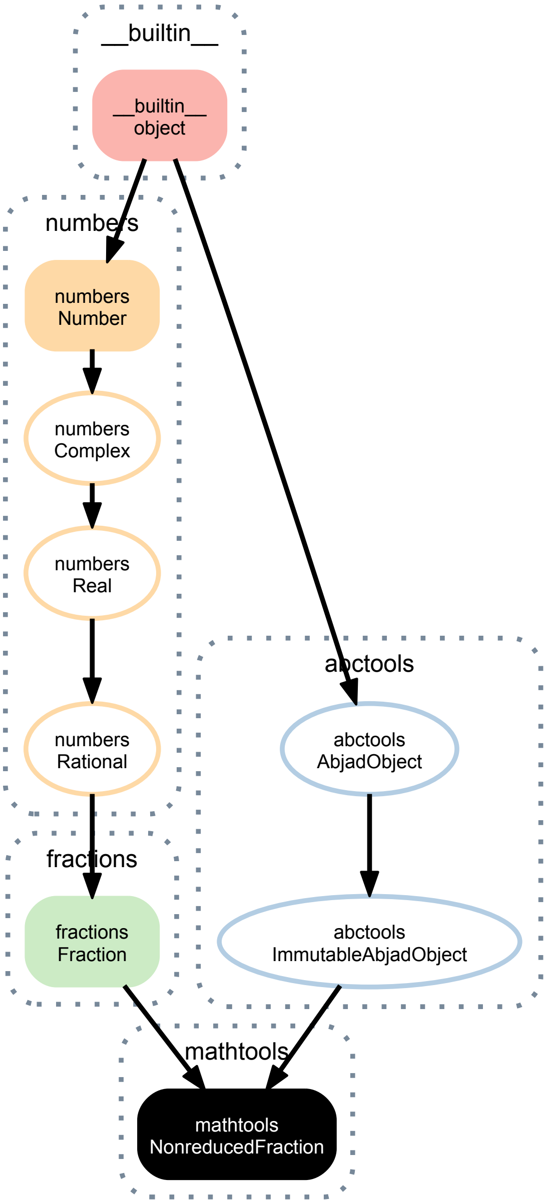
NegativeInfinity.**__ne__**(*expr*)

Defined equal to the opposite of equality.

Return boolean.

NegativeInfinity.**__repr__**()

17.2.3 mathtools.NonreducedFraction



class `mathtools.`**`NonreducedFraction`**(**args*, ***kwargs*)

New in version 2.9. Initialize with an integer numerator and integer denominator:

```
>>> mathtools.NonreducedFraction(3, 6)
NonreducedFraction(3, 6)
```

Initialize with only an integer denominator:

```
>>> mathtools.NonreducedFraction(3)
NonreducedFraction(3, 1)
```

Initialize with an integer pair:

```
>>> mathtools.NonreducedFraction((3, 6))
NonreducedFraction(3, 6)
```

Initialize with an integer singleton:

```
>>> mathtools.NonreducedFraction((3,))
NonreducedFraction(3, 1)
```

Similar to built-in fraction except that numerator and denominator do not reduce.

Nonreduced fractions inherit from built-in fraction:

```
>>> isinstance(mathtools.NonreducedFraction(3, 6), Fraction)
True
```

Nonreduced fractions are numbers:

```
>>> import numbers
```

```
>>> isinstance(mathtools.NonreducedFraction(3, 6), numbers.Number)
True
```

Nonreduced fraction initialization requires more function calls that fraction initialization. But nonreduced fraction initialization is reasonably fast anyway:

```
>>> import fractions
>>> iotools.count_function_calls('fractions.Fraction(3, 6)', globals())
13
```

```
>>> iotools.count_function_calls('mathtools.NonreducedFraction(3, 6)', globals())
28
```

Nonreduced fractions are immutable.

Read-only properties

`NonreducedFraction.`**`denominator`**

Read-only denominator of nonreduced fraction:

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.denominator
3
```

Return positive integer.

`NonreducedFraction.`**`imag`**

Nonreduced fractions have no imaginary part:

```
>>> fraction.imag
0
```

Return zero.

`NonreducedFraction.`**`numerator`**

Read-only numerator of nonreduced fraction:

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.numerator
-6
```

Return integer.

`NonreducedFraction.`**`pair`**

Read only pair of nonreduced fraction numerator and denominator:

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.pair
(-6, 3)
```

Return integer pair.

`NonreducedFraction.`**`real`**

Nonreduced fractions are their own real component:

```
>>> fraction.real
NonreducedFraction(-6, 3)
```

Return nonreduced fraction.

`NonreducedFraction.`**`storage_format`**

Nonreduced fraction storage format:

```
>>> z(fraction)
mathtools.NonreducedFraction(-6, 3)
```

Return string.

Methods

`NonreducedFraction.`**`conjugate`**`()`

Conjugate is a no-op for Reals.

classmethod `NonreducedFraction.`**`from_decimal`**`(`*dec*`)`

Converts a finite Decimal instance to a rational number, exactly.

classmethod `NonreducedFraction.`**`from_float`**`(`*f*`)`

Converts a finite float to a rational number, exactly.

Beware that Fraction.from_float(0.3) != Fraction(3, 10).

`NonreducedFraction.`**`limit_denominator`**`(`*max_denominator=1000000*`)`

Closest Fraction to self with denominator at most max_denominator.

```
>>> Fraction('3.141592653589793').limit_denominator(10)
Fraction(22, 7)
>>> Fraction('3.141592653589793').limit_denominator(100)
Fraction(311, 99)
>>> Fraction(4321, 8765).limit_denominator(10000)
Fraction(4321, 8765)
```

`NonreducedFraction.`**`multiply_with_cross_cancelation`**`(`*multiplier*`)`

New in version 2.11. Multiply nonreduced fraction by *expr* with cross-cancelation:

```
>>> mathtools.NonreducedFraction(4, 8).multiply_with_cross_cancelation((2, 3))
NonreducedFraction(4, 12)
```

```
>>> mathtools.NonreducedFraction(4, 8).multiply_with_cross_cancelation((4, 1))
NonreducedFraction(4, 2)
```

```
>>> mathtools.NonreducedFraction(4, 8).multiply_with_cross_cancelation((3, 5))
NonreducedFraction(12, 40)
```

```
>>> mathtools.NonreducedFraction(4, 8).multiply_with_cross_cancelation((6, 5))
NonreducedFraction(12, 20)
```

```
>>> mathtools.NonreducedFraction(5, 6).multiply_with_cross_cancelation((6, 5))
NonreducedFraction(1, 1)
```

Return nonreduced fraction.

`NonreducedFraction.`**`multiply_with_numerator_preservation`**`(`*multiplier*`)`

Multiply nonreduced fraction by *multiplier* with numerator preservation where possible:

```
>>> mathtools.NonreducedFraction(9, 16).multiply_with_numerator_preservation((2, 3))
NonreducedFraction(9, 24)
```

```
>>> mathtools.NonreducedFraction(9, 16).multiply_with_numerator_preservation((1, 2))
NonreducedFraction(9, 32)
```

```
>>> mathtools.NonreducedFraction(9, 16).multiply_with_numerator_preservation((5, 6))
NonreducedFraction(45, 96)
```

```
>>> mathtools.NonreducedFraction(3, 8).multiply_with_numerator_preservation((2, 3))
NonreducedFraction(3, 12)
```

Return nonreduced fraction.

`NonreducedFraction.`**`multiply_without_reducing`**(*expr*)

Multiply nonreduced fraction by *expr* without reducing:

```
>>> mathtools.NonreducedFraction(3, 8).multiply_without_reducing((3, 3))
NonreducedFraction(9, 24)
```

```
>>> mathtools.NonreducedFraction(4, 8).multiply_without_reducing((4, 5))
NonreducedFraction(16, 40)
```

```
>>> mathtools.NonreducedFraction(4, 8).multiply_without_reducing((3, 4))
NonreducedFraction(12, 32)
```

Return nonreduced fraction.

`NonreducedFraction.`**`reduce`**()

Reduce nonreduced fraction:

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> fraction.reduce()
Fraction(-2, 1)
```

Return fraction.

`NonreducedFraction.`**`with_denominator`**(*denominator*)

Return new nonreduced fraction with integer *denominator*:

```
>>> mathtools.NonreducedFraction(3, 6).with_denominator(12)
NonreducedFraction(6, 12)
```

Return nonreduced fraction.

`NonreducedFraction.`**`with_multiple_of_denominator`**(*denominator*)

Return new nonreduced fraction with multiple of integer *denominator*:

```
>>> mathtools.NonreducedFraction(3, 6).with_multiple_of_denominator(5)
NonreducedFraction(5, 10)
```

Return nonreduced fraction.

Special methods

`NonreducedFraction.`**`__abs__`**()

Absolute value of nonreduced fraction:

```
>>> abs(mathtools.NonreducedFraction(-3, 3))
NonreducedFraction(3, 3)
```

Return nonreduced fraction.

`NonreducedFraction.`**`__add__`**(*expr*)

Add *expr* to nonreduced fraction:

```
>>> mathtools.NonreducedFraction(3, 3) + 1
NonreducedFraction(6, 3)
```

Adding two nonreduced fractions is fairly fast:

```
>>> a = mathtools.NonreducedFraction(3, 6)
>>> b = mathtools.NonreducedFraction(3, 12)
>>> iotools.count_function_calls('a + b', globals())
65
```

Adding an integer is even faster:

```
>>> iotools.count_function_calls('a + 10', globals())
33
```

Return nonreduced fraction.

NonreducedFraction.**__complex__**()
complex(self) == complex(float(self), 0)

NonreducedFraction.**__copy__**()

NonreducedFraction.**__deepcopy__**(*memo*)

NonreducedFraction.**__div__**(*expr*)
Divide nonreduced fraction by expr:

```
>>> mathtools.NonreducedFraction(3, 3) / 1
NonreducedFraction(3, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__divmod__**(*other*)
divmod(self, other): The pair (self // other, self % other).

Sometimes this can be computed faster than the pair of operations.

NonreducedFraction.**__eq__**(*expr*)
True when *expr* equals *self*:

```
>>> mathtools.NonreducedFraction(3, 3) == 1
True
```

Return boolean.

NonreducedFraction.**__float__**()
float(self) = self.numerator / self.denominator

It's important that this conversion use the integer's "true" division rather than casting one side to float before dividing so that ratios of huge integers convert without overflowing.

NonreducedFraction.**__floordiv__**(*a*, *b*)
a // b

NonreducedFraction.**__ge__**(*expr*)
True when nonreduced fraction is greater than or equal to *expr*:

```
>>> mathtools.NonreducedFraction(3, 3) >= 1
True
```

Return boolean.

NonreducedFraction.**__gt__**(*expr*)
True when nonreduced fraction is greater than *expr*:

```
>>> mathtools.NonreducedFraction(3, 3) > 1
False
```

Return boolean.

NonreducedFraction.**__hash__**()
hash(self)

Tricky because values that are exactly representable as a float must have the same hash as that float.

NonreducedFraction.**__le__**(*expr*)
True when nonreduced fraction is less than or equal to *expr*:

```
>>> mathtools.NonreducedFraction(3, 3) <= 1
True
```

Return boolean.

NonreducedFraction.**__lt__**(*expr*)
True when nonreduced fraction is less than *expr*:

```
>>> mathtools.NonreducedFraction(3, 3) < 1
False
```

Return boolean.

NonreducedFraction.**__mod__**(*a*, *b*)
a % b

NonreducedFraction.**__mul__**(*expr*)
Multiply nonreduced fraction by expr:

```
>>> mathtools.NonreducedFraction(3, 3) * 3
NonreducedFraction(9, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__ne__**(*expr*)
True when *expr* does not equal *self*:

```
>>> mathtools.NonreducedFraction(3, 3) != 'foo'
True
```

Return boolean.

NonreducedFraction.**__neg__**()
Negate nonreduced fraction:

```
>>> -mathtools.NonreducedFraction(3, 3)
NonreducedFraction(-3, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__nonzero__**(*a*)
a != 0

NonreducedFraction.**__pos__**(*a*)
+a: Coerces a subclass instance to Fraction

NonreducedFraction.**__pow__**(*a*, *b*)
a ** b

If b is not an integer, the result will be a float or complex since roots are generally irrational. If b is an integer, the result will be rational.

NonreducedFraction.**__radd__**(*expr*)
Add nonreduced fraction to *expr*:

```
>>> 1 + mathtools.NonreducedFraction(3, 3)
NonreducedFraction(6, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__rdiv__**(*expr*)
Divide *expr* by nonreduced fraction:

```
>>> 1 / mathtools.NonreducedFraction(3, 3)
NonreducedFraction(3, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__rdivmod__**(*other*)
divmod(other, self): The pair (self // other, self % other).

Sometimes this can be computed faster than the pair of operations.

NonreducedFraction.**__repr__**()
Interpreter representation of Abjad object.

Return string.

NonreducedFraction.**__rfloordiv__**(*b*, *a*)
a // b

NonreducedFraction.**__rmod__**(*b*, *a*)
a % b

NonreducedFraction.**__rmul__**(*expr*)
Multiply *expr* by nonreduced fraction:

```
>>> 3 * mathtools.NonreducedFraction(3, 3)
NonreducedFraction(9, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__rpow__**(*b*, *a*)
a ** b

NonreducedFraction.**__rsub__**(*expr*)
Subtract nonreduced fraction from *expr*:

```
>>> 1 - mathtools.NonreducedFraction(3, 3)
NonreducedFraction(0, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__rtruediv__**(*b*, *a*)
a / b

NonreducedFraction.**__str__**()
String representation of nonreduced fraction:

```
>>> fraction = mathtools.NonreducedFraction(-6, 3)
```

```
>>> str(fraction)
'-6/3'
```

Return string.

NonreducedFraction.**__sub__**(*expr*)
Subtract *expr* from self:

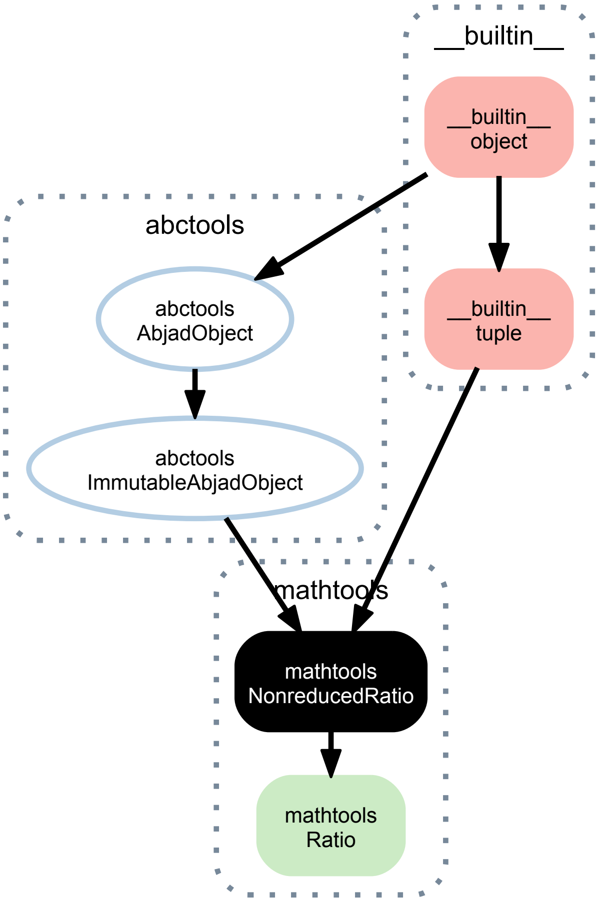
```
>>> mathtools.NonreducedFraction(3, 3) - 2
NonreducedFraction(-3, 3)
```

Return nonreduced fraction.

NonreducedFraction.**__truediv__**(*a*, *b*)
a / b

NonreducedFraction.**__trunc__**(*a*)
trunc(a)

17.2.4 mathtools.NonreducedRatio



class mathtools.**NonreducedRatio** (**args*, ***kwargs*)

New in version 2.11. Nonreduced ratio of one or more nonzero integers.

Initialize from one or more nonzero integers:

```
>>> mathtools.NonreducedRatio(2, 4, 2)
NonreducedRatio(2, 4, 2)
```

Or initialize from a tuple or list:

```
>>> ratio = mathtools.NonreducedRatio((2, 4, 2))
>>> ratio
NonreducedRatio(2, 4, 2)
```

Use a tuple to return ratio integers.

```
>>> tuple(ratio)
(2, 4, 2)
```

Nonreduced ratios are immutable.

Read-only properties

NonreducedRatio.**storage_format**

Storage format of Abjad object.

Return string.

Methods

`NonreducedRatio.count(value)` → integer – return number of occurrences of value

`NonreducedRatio.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`NonreducedRatio.__add__()`
x.__add__(y) <==> x+y

`NonreducedRatio.__contains__()`
x.__contains__(y) <==> y in x

`NonreducedRatio.__eq__(expr)`

`NonreducedRatio.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NonreducedRatio.__getitem__()`
x.__getitem__(y) <==> x[y]

`NonreducedRatio.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`NonreducedRatio.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`NonreducedRatio.__hash__()` <==> *hash(x)*

`NonreducedRatio.__iter__()` <==> *iter(x)*

`NonreducedRatio.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NonreducedRatio.__len__()` <==> *len(x)*

`NonreducedRatio.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NonreducedRatio.__mul__()`
x.__mul__(n) <==> x*n

`NonreducedRatio.__ne__(expr)`
Defined equal to the opposite of equality.

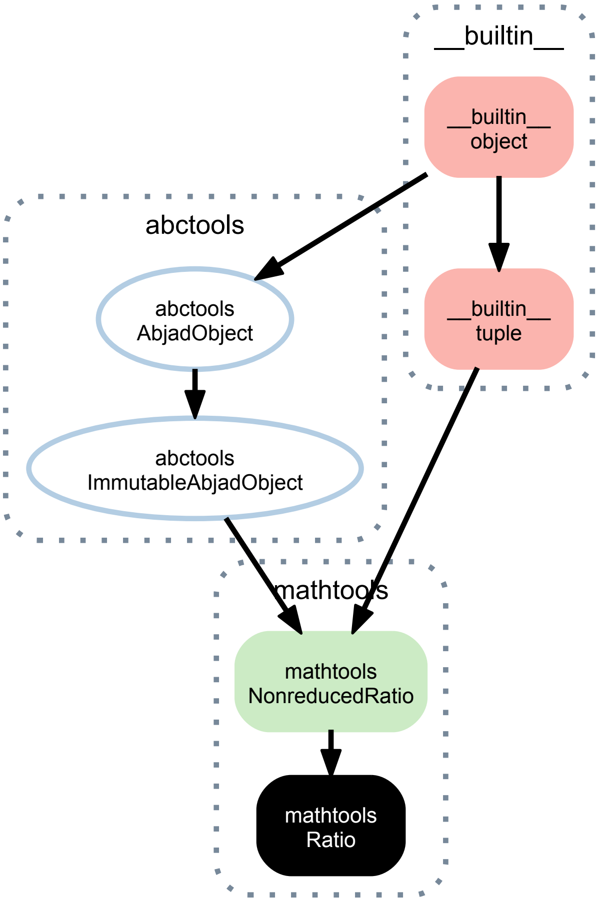
Return boolean.

`NonreducedRatio.__repr__()`
Interpreter representation of Abjad object.

Return string.

`NonreducedRatio.__rmul__()`
x.__rmul__(n) <==> n*x

17.2.5 mathtools.Ratio



class `mathtools.Ratio`(**args*, ***kwargs*)

New in version 2.10. Ratio of one or more nonzero integers.

Initialize from one or more nonzero integers:

```
>>> mathtools.Ratio(2, 4, 2)
Ratio(1, 2, 1)
```

Or initialize from a tuple or list:

```
>>> ratio = mathtools.Ratio((2, 4, 2))
>>> ratio
Ratio(1, 2, 1)
```

Use a tuple to return ratio integers.

```
>>> tuple(ratio)
(1, 2, 1)
```

Ratios are immutable.

Read-only properties

`Ratio.storage_format`

Storage format of Abjad object.

Return string.

Methods

`Ratio.count(value)` → integer – return number of occurrences of value

`Ratio.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`Ratio.__add__()`
x.__add__(y) <==> x+y

`Ratio.__contains__()`
x.__contains__(y) <==> y in x

`Ratio.__eq__(expr)`

`Ratio.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Ratio.__getitem__()`
x.__getitem__(y) <==> x[y]

`Ratio.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`Ratio.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Ratio.__hash__()` <==> *hash(x)*

`Ratio.__iter__()` <==> *iter(x)*

`Ratio.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Ratio.__len__()` <==> *len(x)*

`Ratio.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Ratio.__mul__()`
x.__mul__(n) <==> x*n

`Ratio.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`Ratio.__repr__()`
Interpreter representation of Abjad object.

Return string.

`Ratio.__rmul__()`
x.__rmul__(n) <==> n*x

17.3 Functions

17.3.1 mathtools.are_relatively_prime

`mathtools.are_relatively_prime(expr)`

New in version 2.5. True when *expr* is a sequence comprising zero or more numbers, all of which are relatively prime:

```
>>> mathtools.are_relatively_prime([13, 14, 15])
True
```

Otherwise false:

```
>>> mathtools.are_relatively_prime([13, 14, 15, 16])
False
```

Note that function returns true when *expr* is an empty sequence:

```
>>> mathtools.are_relatively_prime([])
True
```

Function returns false when *expr* is nonsensical type:

```
>>> mathtools.are_relatively_prime('foo')
False
```

Return boolean.

17.3.2 mathtools.arithmetic_mean

`mathtools.arithmetic_mean(sequence)`

New in version 1.1. Arithmetic means of *sequence* as an exact integer:

```
>>> mathtools.arithmetic_mean([1, 2, 2, 20, 30])
11
```

As a rational:

```
>>> mathtools.arithmetic_mean([1, 2, 20])
Fraction(23, 3)
```

As a float:

```
>>> mathtools.arithmetic_mean([2, 2, 20.0])
8.0
```

Return number.

17.3.3 mathtools.binomial_coefficient

`mathtools.binomial_coefficient(n, k)`

New in version 2.0. Binomial coefficient of *n* choose *k*:

```
>>> for k in range(8):
...     print k, '\t', mathtools.binomial_coefficient(8, k)
...
0   1
1   8
2   28
3   56
4   70
5   56
6   28
7   8
```

Return positive integer.

17.3.4 mathtools.cumulative_products

`mathtools.cumulative_products(sequence)`

Cumulative products of *sequence*:

```
>>> mathtools.cumulative_products([1, 2, 3, 4, 5, 6, 7, 8])
[1, 2, 6, 24, 120, 720, 5040, 40320]
```

```
>>> mathtools.cumulative_products([1, -2, 3, -4, 5, -6, 7, -8])
[1, -2, -6, 24, 120, -720, -5040, 40320]
```

Raise type error when *sequence* is neither list nor tuple.

Raise value error on empty *sequence*.

Return list.

17.3.5 mathtools.cumulative_signed_weights

`mathtools.cumulative_signed_weights(sequence)`

Cumulative signed weights of *sequence*:

```
>>> l = [1, -2, -3, 4, -5, -6, 7, -8, -9, 10]
>>> mathtools.cumulative_signed_weights(l)
[1, -3, -6, 10, -15, -21, 28, -36, -45, 55]
```

Raise type error when *sequence* is not a list.

For cumulative (unsigned) weights use `mathtools.cumulative_sums([abs(x) for x in l])`.

Return list.

17.3.6 mathtools.cumulative_sums

`mathtools.cumulative_sums(sequence)`

Cumulative sums of *sequence*:

```
>>> mathtools.cumulative_sums([1, 2, 3, 4, 5, 6, 7, 8])
[1, 3, 6, 10, 15, 21, 28, 36]
```

Raise type error when *sequence* is neither list nor tuple.

Raise value error on empty *sequence*.

Return list.

17.3.7 mathtools.cumulative_sums_zero

`mathtools.cumulative_sums_zero(sequence)`

Cumulative sums of *sequence* starting from `0`:

```
>>> mathtools.cumulative_sums_zero([1, 2, 3, 4, 5, 6, 7, 8])
[0, 1, 3, 6, 10, 15, 21, 28, 36]
```

Return `[0]` on empty *sequence*:

```
>>> mathtools.cumulative_sums_zero([])
[0]
```

Return list.

17.3.8 mathtools.cumulative_sums_zero_pairwise

`mathtools.cumulative_sums_zero_pairwise(sequence)`

List pairwise cumulative sums of *sequence* from 0:

```
>>> mathtools.cumulative_sums_zero_pairwise([1, 2, 3, 4, 5, 6])
[(0, 1), (1, 3), (3, 6), (6, 10), (10, 15), (15, 21)]
```

Return list of pairs.

17.3.9 mathtools.difference_series

`mathtools.difference_series(sequence)`

Difference series of *sequence*:

```
>>> mathtools.difference_series([1, 1, 2, 3, 5, 5, 6])
[0, 1, 1, 2, 0, 1]
```

Return list.

17.3.10 mathtools.divide_number_by_ratio

`mathtools.divide_number_by_ratio(number, ratio)`

Divide integer by *ratio*:

```
>>> mathtools.divide_number_by_ratio(1, [1, 1, 3])
[Fraction(1, 5), Fraction(1, 5), Fraction(3, 5)]
```

Divide fraction by *ratio*:

```
>>> mathtools.divide_number_by_ratio(Fraction(1), [1, 1, 3])
[Fraction(1, 5), Fraction(1, 5), Fraction(3, 5)]
```

Divide float by ratio:

```
>>> mathtools.divide_number_by_ratio(1.0, [1, 1, 3])
[0.20000000000000001, 0.20000000000000001, 0.60000000000000009]
```

Raise type error on nonnumeric *number*.

Raise type error on noninteger in *ratio*.

Return list of fractions or list of floats.

17.3.11 mathtools.divisors

`mathtools.divisors(n)`

Positive divisors of integer *n* in increasing order:

```
>>> mathtools.divisors(84)
[1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42, 84]
```

```
>>> for x in range(10, 20):
...     print x, mathtools.divisors(x)
...
10 [1, 2, 5, 10]
11 [1, 11]
12 [1, 2, 3, 4, 6, 12]
13 [1, 13]
14 [1, 2, 7, 14]
15 [1, 3, 5, 15]
16 [1, 2, 4, 8, 16]
17 [1, 17]
18 [1, 2, 3, 6, 9, 18]
19 [1, 19]
```

Allow nonpositive *n*:

```
>>> mathtools.divisors(-27)
[1, 3, 9, 27]
```

Performance is extremely fast:

```
>>> iotools.count_function_calls('mathtools.divisors(100000000)', globals())
50
```

Raise type error on noninteger *n*.

Raise not implemented error on `0`.

Return list of positive integers.

17.3.12 mathtools.factors

mathtools.**factors**(*n*)

Integer factors of positive integer *n* in increasing order:

```
>>> mathtools.factors(84)
[1, 2, 2, 3, 7]
```

```
>>> for n in range(10, 20):
...   print n, mathtools.factors(n)
...
10 [1, 2, 5]
11 [1, 11]
12 [1, 2, 2, 3]
13 [1, 13]
14 [1, 2, 7]
15 [1, 3, 5]
16 [1, 2, 2, 2, 2]
17 [1, 17]
18 [1, 2, 3, 3]
19 [1, 19]
```

Raise type error on noninteger *n*.

Raise value error on nonpositive *n*.

Return list of one or more positive integers.

17.3.13 mathtools.fraction_to_proper_fraction

mathtools.**fraction_to_proper_fraction**(*rational*)

New in version 2.0. Change *rational* to proper fraction:

```
>>> mathtools.fraction_to_proper_fraction(Fraction(116, 8))
(14, Fraction(1, 2))
```

Return pair.

17.3.14 mathtools.get_shared_numeric_sign

mathtools.**get_shared_numeric_sign**(*sequence*)

Return `1` when all *sequence* elements are positive:

```
>>> mathtools.get_shared_numeric_sign([1, 2, 3])
1
```

Return `-1` when all *sequence* elements are negative:

```
>>> mathtools.get_shared_numeric_sign([-1, -2, -3])
-1
```

Return `0` on empty *sequence*:

```
>>> mathtools.get_shared_numeric_sign([])
0
```

Otherwise return none:

```
>>> mathtools.get_shared_numeric_sign([1, 2, -3]) is None
True
```

Return `1`, `-1`, `0` or none.

17.3.15 mathtools.greatest_common_divisor

`mathtools.greatest_common_divisor(*integers)`

New in version 2.0. Calculate greatest common divisor of *integers*:

```
>>> mathtools.greatest_common_divisor(84, -94, -144)
2
```

Allow nonpositive input.

Raise type error on noninteger input.

Raise not implemented error when `0` is included in input.

Return positive integer.

17.3.16 mathtools.greatest_multiple_less_equal

`mathtools.greatest_multiple_less_equal(m, n)`

Greatest integer multiple of *m* less than or equal to *n*:

```
>>> mathtools.greatest_multiple_less_equal(10, 47)
40
```

```
>>> for m in range(1, 10):
...     print m, mathtools.greatest_multiple_less_equal(m, 47)
...
1 47
2 46
3 45
4 44
5 45
6 42
7 42
8 40
9 45
```

```
>>> for n in range(10, 100, 10):
...     print mathtools.greatest_multiple_less_equal(7, n), n
...
7 10
14 20
28 30
35 40
49 50
56 60
70 70
77 80
84 90
```

Raise type error on nonnumeric *m*.

Raise type error on nonnumeric *n*.

Return nonnegative integer.

17.3.17 mathtools.greatest_power_of_two_less_equal

mathtools.**greatest_power_of_two_less_equal**(*n*, *i=0*)

Greatest integer power of two less than or equal to positive *n*:

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.greatest_power_of_two_less_equal(n))
...
    10  8
    11  8
    12  8
    13  8
    14  8
    15  8
    16  16
    17  16
    18  16
    19  16
```

Greatest-but-i integer power of 2 less than or equal to positive *n*:

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.greatest_power_of_two_less_equal(n, i=1))
...
    10  4
    11  4
    12  4
    13  4
    14  4
    15  4
    16  8
    17  8
    18  8
    19  8
```

Raise type error on nonnumeric *n*.

Raise value error on nonpositive *n*.

Return positive integer.

17.3.18 mathtools.integer_equivalent_number_to_integer

mathtools.**integer_equivalent_number_to_integer**(*number*)

New in version 2.0. Integer-equivalent *number* to integer:

```
>>> mathtools.integer_equivalent_number_to_integer(17.0)
17
```

Return noninteger-equivalent number unchanged:

```
>>> mathtools.integer_equivalent_number_to_integer(17.5)
17.5
```

Raise type error on nonnumber input.

Return number.

17.3.19 mathtools.integer_to_base_k_tuple

mathtools.**integer_to_base_k_tuple**(*n*, *k*)

New in version 2.0. Nonnegative integer *n* to base-*k* tuple:

```
>>> mathtools.integer_to_base_k_tuple(1066, 10)
(1, 0, 6, 6)
```

Return tuple of one or more positive integers.

17.3.20 mathtools.integer_to_binary_string

mathtools.**integer_to_binary_string**(*n*)

Positive integer *n* to binary string:

```
>>> for n in range(1, 16 + 1):
...     print '{}\t{}'.format(n, mathtools.integer_to_binary_string(n))
...
1  1
2  10
3  11
4  100
5  101
6  110
7  111
8  1000
9  1001
10 1010
11 1011
12 1100
13 1101
14 1110
15 1111
16 10000
```

Return string.

17.3.21 mathtools.interpolate_cosine

mathtools.**interpolate_cosine**(*y1*, *y2*, *mu*)

Cosine interpolate *y1* and *y2* with *mu* normalized `[0, 1]`:

```
>>> mathtools.interpolate_cosine(0, 1, 0.5)
0.49999999999999994
```

Return float.

17.3.22 mathtools.interpolate_divide

mathtools.**interpolate_divide**(*total*, *start_fraction*, *stop_fraction*, *exp='cosine'*)

Divide *total* into segments of sizes computed from interpolating between *start_fraction* and *stop_fraction*:

```
>>> mathtools.interpolate_divide(10, 1, 1, exp=1)
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> sum(mathtools.interpolate_divide(10, 1, 1, exp=1))
10.0
```

```
>>> mathtools.interpolate_divide(10, 5, 1)
[4.7986734489043181, 2.8792040693425909, 1.3263207210948171,
0.99580176065827419]
>>> sum(mathtools.interpolate_divide(10, 5, 1))
10.0
```

Set `exp='cosine'` for cosine interpolation.

Set *exp* to a numeric value for exponential interpolation with *exp* as the exponent.

Scale resulting segments so that their sum equals exactly *total*.

Return a list of floats.

17.3.23 mathtools.interpolate_divide_multiple

`mathtools.interpolate_divide_multiple(totals, key_values, exp='cosine')`

New in version 2.0. Interpolate *key_values* such that the sum of the resulting interpolated values equals the given *totals*:

```
>>> mathtools.interpolate_divide_multiple([100, 50], [20, 10, 20])
[19.4487, 18.5201, 16.2270, 13.7156, 11.7488, 10.4879,
9.8515, 9.5130, 10.4213, 13.0736, 16.9918]
```

The operation is the same as `mathtools.interpolate_divide()`. But this function takes multiple *totals* and *key_values* at once.

Precondition: `len(totals) == len(key_values) - 1`.

Set *totals* equal to a list or tuple of the total sum of interpolated values.

Set *key_values* equal a list or tuple of key values to interpolate.

Set *exp* to *consine* for consine interpolation.

Set *exp* to a number for exponential interpolation.

Returns a list of floats.

17.3.24 mathtools.interpolate_exponential

`mathtools.interpolate_exponential(y1, y2, mu, exp=1)`

Exponential interpolate *y1* and *y2* with *mu* normalized `[0, 1]`:

```
>>> mathtools.interpolate_exponential(0, 1, 0.5, 4)
0.0625
```

Set *exp* equal to the exponent of interpolation.

Return float.

17.3.25 mathtools.interpolate_linear

`mathtools.interpolate_linear(y1, y2, mu)`

Linear interpolate *y1* and *y2* with *mu* normalized `[0, 1]`:

```
>>> mathtools.interpolate_linear(0, 1, 0.5)
0.5
```

Return float.

17.3.26 mathtools.interval_string_to_pair_and_indicators

`mathtools.interval_string_to_pair_and_indicators(interval_string)`

New in version 1.0. Change *interval_string* to pair, boolean start indicator and boolean stop indicator:

```
>>> mathtools.interval_string_to_pair_and_indicators('[5, 8)')
((5, 8), False, True)
```

Parse square brackets as closed interval bounds.

Parse parentheses as open interval bounds.

Return triple.

17.3.27 mathtools.is_assignable_integer

`mathtools.is_assignable_integer(expr)`
New in version 2.0. True when *expr* is equivalent to an integer and can be written without recourse to ties:

```
>>> for n in range(0, 16 + 1):
...     print '%s\t%s' % (n, mathtools.is_assignable_integer(n))
...
0   False
1   True
2   True
3   True
4   True
5   False
6   True
7   True
8   True
9   False
10  False
11  False
12  True
13  False
14  True
15  True
16  True
```

Otherwise false.

Return boolean.

17.3.28 mathtools.is_dotted_integer

`mathtools.is_dotted_integer(expr)`
New in version 2.0. True when *expr* is equivalent to a positive integer and can be written with zero or more dots:

```
>>> for expr in range(16):
...     print '%s        %s' % (expr, mathtools.is_dotted_integer(expr))
...
0         False
1         False
2         False
3         True
4         False
5         False
6         True
7         True
8         False
9         False
10        False
11        False
12        True
13        False
14        True
15        True
```

Otherwise false.

Return boolean.

Integer *n* qualifies as dotted when `abs(n)` is of the form `2**j * (2**k - 1)` with integers `0 <= j`, `2 < k`.

17.3.29 mathtools.is_integer_equivalent_expr

`mathtools.is_integer_equivalent_expr(expr)`
New in version 2.5. True when *expr* is an integer-equivalent number:

```
>>> mathtools.is_integer_equivalent_expr(12.0)
True
```

True when *expr* evaluates to an integer:

```
>>> mathtools.is_integer_equivalent_expr('12')
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_expr('foo')
False
```

Return boolean.

17.3.30 mathtools.is_integer_equivalent_number

mathtools.**is_integer_equivalent_number**(*expr*)
New in version 2.0. True when *expr* is a number and *expr* is equivalent to an integer:

```
>>> mathtools.is_integer_equivalent_number(12.0)
True
```

Otherwise false:

```
>>> mathtools.is_integer_equivalent_number(Duration(1, 2))
False
```

Return boolean.

17.3.31 mathtools.is_negative_integer

mathtools.**is_negative_integer**(*expr*)
New in version 2.0. True when *expr* equals a negative integer:

```
>>> mathtools.is_negative_integer(-1)
True
```

Otherwise false:

```
>>> mathtools.is_negative_integer(0)
False
```

```
>>> mathtools.is_negative_integer(99)
False
```

Return boolean.

17.3.32 mathtools.is_nonnegative_integer

mathtools.**is_nonnegative_integer**(*expr*)
New in version 2.0. True when *expr* equals a nonnegative integer:

```
>>> mathtools.is_nonnegative_integer(99)
True
```

```
>>> mathtools.is_nonnegative_integer(0)
True
```

Otherwise false:

```
>>> mathtools.is_nonnegative_integer(-1)
False
```

Return boolean.

17.3.33 mathtools.is_nonnegative_integer_equivalent_number

mathtools.**is_nonnegative_integer_equivalent_number**(*expr*)

New in version 2.0. True when *expr* is a nonnegative integer-equivalent number. Otherwise false:

```
>>> mathtools.is_nonnegative_integer_equivalent_number(Duration(4, 2))
True
```

Return boolean.

17.3.34 mathtools.is_nonnegative_integer_power_of_two

mathtools.**is_nonnegative_integer_power_of_two**(*expr*)

True when *expr* is a nonnegative integer power of 2:

```
>>> for n in range(10):
...     print n, mathtools.is_nonnegative_integer_power_of_two(n)
...
0 True
1 True
2 True
3 False
4 True
5 False
6 False
7 False
8 True
9 False
```

Otherwise false.

Return boolean.

17.3.35 mathtools.is_positive_integer

mathtools.**is_positive_integer**(*expr*)

New in version 2.0. True when *expr* equals a positive integer:

```
>>> mathtools.is_positive_integer(99)
True
```

Otherwise false:

```
>>> mathtools.is_positive_integer(0)
False
```

```
>>> mathtools.is_positive_integer(-1)
False
```

Return boolean.

17.3.36 mathtools.is_positive_integer_equivalent_number

mathtools.**is_positive_integer_equivalent_number**(*expr*)

New in version 2.0. True when *expr* is a positive integer-equivalent number. Otherwise false:

```
>>> mathtools.is_positive_integer_equivalent_number(Duration(4, 2))
True
```

Return boolean.

17.3.37 mathtools.is_positive_integer_power_of_two

`mathtools.`**`is_positive_integer_power_of_two`**`(`*expr*`)`

True when *expr* is a positive integer power of `2`:

```
>>> for n in range(10):
...     print n, mathtools.is_positive_integer_power_of_two(n)
...
0 False
1 True
2 True
3 False
4 True
5 False
6 False
7 False
8 True
9 False
```

Otherwise false.

Return boolean.

17.3.38 mathtools.least_common_multiple

`mathtools.`**`least_common_multiple`**`(`**integers*`)`

Least common multiple of positive *integers*:

```
>>> mathtools.least_common_multiple(2, 4, 5, 10, 20)
20
```

Return positive integer.

17.3.39 mathtools.least_multiple_greater_equal

`mathtools.`**`least_multiple_greater_equal`**`(`*m*, *n*`)`

Return the least integer multiple of *m* greater than or equal to *n*:

```
>>> mathtools.least_multiple_greater_equal(10, 47)
50
```

```
>>> for m in range(1, 10):
...     print m, mathtools.least_multiple_greater_equal(m, 47)
...
1 47
2 48
3 48
4 48
5 50
6 48
7 49
8 48
9 54
```

```
>>> for n in range(10, 100, 10):
...     print mathtools.least_multiple_greater_equal(7, n), n
...
14 10
21 20
35 30
42 40
56 50
63 60
70 70
84 80
91 90
```

Return integer.

17.3.40 mathtools.least_power_of_two_greater_equal

mathtools.**least_power_of_two_greater_equal**(*n*, *i=0*)

Return least integer power of two greater than or equal to positive *n*:

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.least_power_of_two_greater_equal(n))
...
    10 16
    11 16
    12 16
    13 16
    14 16
    15 16
    16 16
    17 32
    18 32
    19 32
```

When `i = 1`, return the first integer power of `2` greater than the least integer power of `2` greater than or equal to *n*.

```
>>> for n in range(10, 20):
...     print '\t%s\t%s' % (n, mathtools.least_power_of_two_greater_equal(n, i=1))
...
    10 32
    11 32
    12 32
    13 32
    14 32
    15 32
    16 32
    17 64
    18 64
    19 64
```

When `i = 2`, return the second integer power of `2` greater than the least integer power of `2` greater than or equal to *n*, and, in general, return the `i` th integer power of `2` greater than the least integer power of `2` greater than or equal to *n*.

Raise type error on nonnumeric *n*.

Raise value error on nonpositive *n*.

Return integer.

17.3.41 mathtools.next_integer_partition

mathtools.**next_integer_partition**(*integer_partition*)

New in version 2.0. Next integer partition following *integer_partition* in descending lex order:

```
>>> mathtools.next_integer_partition((8, 3))
(8, 2, 1)
```

```
>>> mathtools.next_integer_partition((8, 2, 1))
(8, 1, 1, 1)
```

```
>>> mathtools.next_integer_partition((8, 1, 1, 1))
(7, 4)
```

Input *integer_partition* must be sequence of positive integers.

Return integer partition as tuple of positive integers.

17.3.42 mathtools.partition_integer_by_ratio

mathtools.**partition_integer_by_ratio**(*n*, *ratio*)

Partition positive integer-equivalent *n* by *ratio*:

```
>>> mathtools.partition_integer_by_ratio(10, [1, 2])
[3, 7]
```

Partition positive integer-equivalent *n* by *ratio* with negative parts:

```
>>> mathtools.partition_integer_by_ratio(10, [1, -2])
[3, -7]
```

Partition negative integer-equivalent *n* by *ratio*:

```
>>> mathtools.partition_integer_by_ratio(-10, [1, 2])
[-3, -7]
```

Partition negative integer-equivalent *n* by *ratio* with negative parts:

```
>>> mathtools.partition_integer_by_ratio(-10, [1, -2])
[-3, 7]
```

Return result with weight equal to absolute value of *n*.

Raise type error on noninteger *n*.

Return list of integers.

17.3.43 mathtools.partition_integer_into_canonic_parts

mathtools.**partition_integer_into_canonic_parts**(*n*, *decrease_parts_monotonically=True*)

Partition integer *n* into canonic parts.

Return all parts positive on positive *n*:

```
>>> for n in range(1, 11):
...     print n, mathtools.partition_integer_into_canonic_parts(n)
...
1 (1,)
2 (2,)
3 (3,)
4 (4,)
5 (4, 1)
6 (6,)
7 (7,)
8 (8,)
9 (8, 1)
10 (8, 2)
```

Return all parts negative on negative *n*:

```
>>> for n in reversed(range(-20, -10)):
...     print n, mathtools.partition_integer_into_canonic_parts(n)
...
-11 (-8, -3)
-12 (-12,)
-13 (-12, -1)
-14 (-14,)
-15 (-15,)
-16 (-16,)
-17 (-16, -1)
-18 (-16, -2)
-19 (-16, -3)
-20 (-16, -4)
```

Return parts that increase monotonically:

```
>>> for n in range(11, 21):
...     print n, mathtools.partition_integer_into_canonic_parts(n,
...         decrease_parts_monotonically=False)
...
11 (3, 8)
12 (12,)
13 (1, 12)
14 (14,)
15 (15,)
16 (16,)
17 (1, 16)
18 (2, 16)
19 (3, 16)
20 (4, 16)
```

Return tuple with parts that decrease monotonically.

Raise type error on noninteger *n*.

Return tuple of one or more integers.

17.3.44 mathtools.partition_integer_into_halves

mathtools.**partition_integer_into_halves**(*n*, *bigger='left'*, *even='allowed'*)

Write positive integer *n* as the pair `t = (left, right)` such that `n == left + right`.

When *n* is odd the greater part of `t` corresponds to the value of *bigger*:

```
>>> mathtools.partition_integer_into_halves(7, bigger='left')
(4, 3)
>>> mathtools.partition_integer_into_halves(7, bigger='right')
(3, 4)
```

Likewise when *n* is even and `even = 'disallowed'`:

```
>>> mathtools.partition_integer_into_halves(8, bigger='left', even='disallowed')
(5, 3)
>>> mathtools.partition_integer_into_halves(8, bigger='right', even='disallowed')
(3, 5)
```

But when *n* is even and `even = 'allowed'` then `left == right` and *bigger* is ignored:

```
>>> mathtools.partition_integer_into_halves(8)
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='left')
(4, 4)
>>> mathtools.partition_integer_into_halves(8, bigger='right')
(4, 4)
```

When *n* is `0` return `(0, 0)`:

```
>>> mathtools.partition_integer_into_halves(0)
(0, 0)
```

When *n* is `0` and `even = 'disallowed'` raise partition error.

Raise type error on noninteger *n*.

Raise value error on negative *n*.

Return pair of positive integers.

17.3.45 mathtools.partition_integer_into_parts_less_than_double

mathtools.**partition_integer_into_parts_less_than_double**(*n*, *m*)

Partition integer *n* into parts less than double integer *m*.

Example:

```
>>> for n in range(1, 24+1):
...     print n, mathtools.partition_integer_into_parts_less_than_double(n, 4)
1 (1,)
2 (2,)
3 (3,)
4 (4,)
5 (5,)
6 (6,)
7 (7,)
8 (4, 4)
9 (4, 5)
10 (4, 6)
11 (4, 7)
12 (4, 4, 4)
13 (4, 4, 5)
14 (4, 4, 6)
15 (4, 4, 7)
16 (4, 4, 4, 4)
17 (4, 4, 4, 5)
18 (4, 4, 4, 6)
19 (4, 4, 4, 7)
20 (4, 4, 4, 4, 4)
21 (4, 4, 4, 4, 5)
22 (4, 4, 4, 4, 6)
23 (4, 4, 4, 4, 7)
24 (4, 4, 4, 4, 4, 4)
```

Return tuple of one or more integers.

17.3.46 mathtools.partition_integer_into_units

mathtools.**partition_integer_into_units**(*n*)

Partition positive integer into units:

```
>>> mathtools.partition_integer_into_units(6)
[1, 1, 1, 1, 1, 1]
```

Partition negative integer into units:

```
>>> mathtools.partition_integer_into_units(-5)
[-1, -1, -1, -1, -1]
```

Partition `0` into units:

```
>>> mathtools.partition_integer_into_units(0)
[]
```

Return list of zero or more parts with absolute value equal to `1`.

17.3.47 mathtools.remove_powers_of_two

mathtools.**remove_powers_of_two**(*n*)

Remove powers of `2` from the factors of positive integer *n*:

```
>>> for n in range(10, 100, 10):
...     print '\t%s\t%s' % (n, mathtools.remove_powers_of_two(n))
...
    10 5
    20 5
    30 15
    40 5
    50 25
    60 15
    70 35
    80 5
    90 45
```

Raise type error on noninteger *n*.

Raise value error on nonpositive *n*.

Return positive integer.

17.3.48 mathtools.sign

`mathtools.sign(n)`

Return `-1` on negative *n*:

```
>>> mathtools.sign(-96.2)
-1
```

Return `0` when *n* is `0`:

```
>>> mathtools.sign(0)
0
```

Return `1` on positive *n*:

```
>>> mathtools.sign(Duration(9, 8))
1
```

Return `-1`, `0` or `1`.

17.3.49 mathtools.weight

`mathtools.weight(sequence)`

Sum of the absolute value of the elements in *sequence*:

```
>>> mathtools.weight([-1, -2, 3, 4, 5])
15
```

Return nonnegative integer.

17.3.50 mathtools.yield_all_compositions_of_integer

`mathtools.yield_all_compositions_of_integer(n)`

New in version 2.0. Yield all compositions of positive integer *n* in descending lex order:

```
>>> for integer_composition in mathtools.yield_all_compositions_of_integer(5):
...     integer_composition
...
(5,)
(4, 1)
(3, 2)
(3, 1, 1)
(2, 3)
(2, 2, 1)
(2, 1, 2)
(2, 1, 1, 1)
(1, 4)
(1, 3, 1)
(1, 2, 2)
(1, 2, 1, 1)
(1, 1, 3)
(1, 1, 2, 1)
(1, 1, 1, 2)
(1, 1, 1, 1, 1)
```

Integer compositions are ordered integer partitions.

Return generator of positive integer tuples of length at least `1`.

17.3.51 mathtools.yield_all_partitions_of_integer

mathtools.**yield_all_partitions_of_integer**(*n*)

New in version 2.0. Yield all partitions of positive integer *n* in descending lex order:

```
>>> for partition in mathtools.yield_all_partitions_of_integer(7):
...     partition
...
(7,)
(6, 1)
(5, 2)
(5, 1, 1)
(4, 3)
(4, 2, 1)
(4, 1, 1, 1)
(3, 3, 1)
(3, 2, 2)
(3, 2, 1, 1)
(3, 1, 1, 1, 1)
(2, 2, 2, 1)
(2, 2, 1, 1, 1)
(2, 1, 1, 1, 1, 1)
(1, 1, 1, 1, 1, 1, 1)
```

Return generator of positive integer tuples of length at least `1`.

17.3.52 mathtools.yield_nonreduced_fractions

mathtools.**yield_nonreduced_fractions**()

New in version 2.0. Yield positive nonreduced fractions in Cantor diagonalized order:

```
>>> generator = mathtools.yield_nonreduced_fractions()
>>> for n in range(16):
...     generator.next()
...
(1, 1)
(2, 1)
(1, 2)
(1, 3)
(2, 2)
(3, 1)
(4, 1)
(3, 2)
(2, 3)
(1, 4)
(1, 5)
(2, 4)
(3, 3)
(4, 2)
(5, 1)
(6, 1)
```

Note: function should return nonreduced fraction generator instead of pair generator.

Return generator.

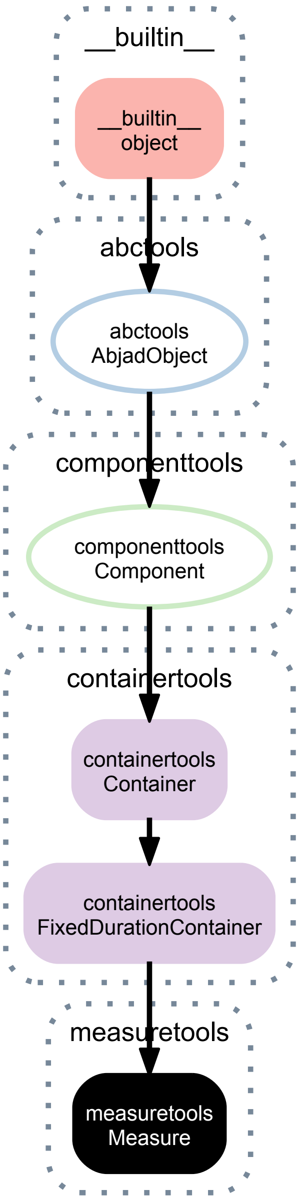
CHAPTER

EIGHTEEN

MEASURETOOLS

18.1 Concrete Classes

18.1.1 measuretools.Measure



class measuretools.**Measure** (*time_signature*, *music=None*, ***kwargs*)

New in version 1.1. Abjad model of a measure:

```
>>> measure = Measure((4, 8), "c'8 d' e' f'")
```

```
>>> measure
Measure(4/8, [c'8, d'8, e'8, f'8])
```

```
>>> f(measure)
{
    \time 4/8
    c'8
    d'8
    e'8
    f'8
}
```

Return measure object.

Read-only properties

Measure.**contents_duration**

Measure.**descendants**

Read-only reference to component descendants score selection.

Measure.**duration**

Measure.**duration_in_seconds**

Measure.**has_non_power_of_two_denominator**

True when measure time signature denominator is not an integer power of 2:

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> measure.has_non_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 8), "c'8 d' e' f' g'")
>>> measure.has_non_power_of_two_denominator
False
```

Return boolean.

Measure.**has_power_of_two_denominator**

True when measure time signature denominator is an integer power of 2:

```
>>> measure = Measure((5, 8), "c'8 d' e' f' g'")
>>> measure.has_power_of_two_denominator
True
```

Otherwise false:

```
>>> measure = Measure((5, 9), "c'8 d' e' f' g'")
>>> measure.has_power_of_two_denominator
False
```

Return boolean.

Measure.**implied_prolation**

Implied prolation of measure time signature:

```
>>> measure = Measure((5, 12), "c'8 d' e' f' g'")
```

```
>>> measure.implied_prolation
Multiplier(2, 3)
```

Return multiplier.

Measure.**is_full**
True when prolated duration equals time signature duration:

```
>>> measure = Measure((4, 8), "c'8 d'8 e'8 f'8")
```

```
>>> measure.is_full
True
```

Otherwise false.

Return boolean.

Measure.**is_misfilled**
New in version 2.9. True when measure is either underfull or overfull:

```
>>> measure = Measure((3, 4), "c' d' e' f'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4, f'4])
```

```
>>> measure.is_misfilled
True
```

Otherwise false:

```
>>> measure = Measure((3, 4), "c' d' e'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.is_misfilled
False
```

Return boolean.

Measure.**is_overfull**
New in version 1.1. True when prolated duration is greater than time signature duration:

```
>>> measure = Measure((3, 4), "c'4 d' e' f'")
```

```
>>> measure.is_overfull
True
```

Otherwise false.

Return boolean.

Measure.**is_underfull**
New in version 1.1. True when prolated duration is less than time signature duration:

```
>>> measure = Measure((3, 4), "c'4 d'")
```

```
>>> measure.is_underfull
True
```

Otherwise false.

Return boolean.

Measure.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

Measure.**lilypond_format**

Measure.**lineage**
Read-only reference to component lineage score selection.

Measure.**measure_number**
1-indexed measure number:

```
>>> staff = Staff()
>>> staff.append(Measure((3, 4), "c' d' e'"))
>>> staff.append(Measure((2, 4), "f' g'"))
```

```
>>> f(staff)
\new Staff {
    {
        \time 3/4
        c'4
        d'4
        e'4
    }
    {
        \time 2/4
        f'4
        g'4
    }
}
```

```
>>> staff[0].measure_number
1
```

```
>>> staff[1].measure_number
2
```

Return positive integer.

Measure.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

Measure.**override**
Read-only reference to LilyPond grob override component plug-in.

Measure.**parent**

Measure.**parentage**
Read-only reference to component parentage score selection.

Measure.**preprolated_duration**
Preprolated duration of measure:

```
>>> measure = Measure((5, 12), "c'8 d' e' f' g'")
```

```
>>> measure.preprolated_duration
Duration(5, 12)
```

Equal measure contents duration times time signature multiplier.

Return duration.

Measure.**prolation**

Measure.**set**
Read-only reference LilyPond context setting component plug-in.

Measure.**spanners**
Read-only reference to unordered set of spanners attached to component.

`Measure.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Measure.`**`target_duration`**
New in version 2.9. Read-only target duration of measure always equal to duration of effective time signature.

`Measure.`**`timespan`**
Read-only timespan of component.

`Measure.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Read/write properties

`Measure.`**`always_format_time_signature`**
New in version 2.9. Read / write flag to indicate whether time signature should appear in LilyPond format even when not expected.

Set to true when necessary to print the same signature repeatedly.

Default to false.

Return boolean.

`Measure.`**`automatically_adjust_time_signature`**
New in version 2.9. Read / write flag to indicate whether time signature should update automatically following contents-changing operations:

```
>>> measure = Measure((3, 4), "c' d' e'")
```

```
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.automatically_adjust_time_signature = True
>>> measure.append('r')
```

```
>>> measure
Measure(4/4, [c'4, d'4, e'4, r4])
```

Default to false.

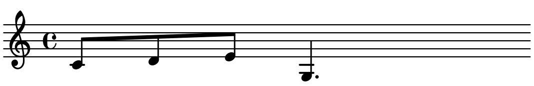
Return boolean.

`Measure.`**`is_parallel`**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

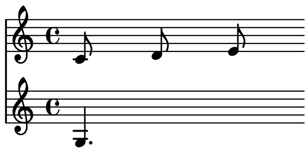
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

Methods

`Measure.`**append**(*component*)

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

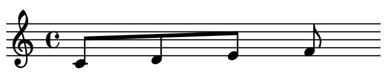
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

Measure.**extend**(*expr*)
Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

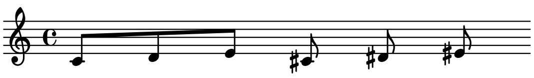
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

Measure.**index**(*component*)
Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

Measure.**insert**(*i*, *component*)
Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

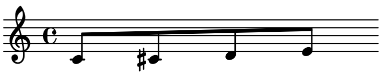
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Measure.pop(i=-1)`
Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

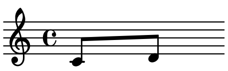
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Measure.remove(component)`
Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

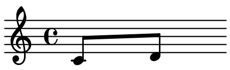
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Measure.__add__(arg)`
Add two measures together in-score or outside-of-score. Wrapper around measuretools.fuse_measures.

`Measure.__contains__(expr)`
True if expr is in container, otherwise False.

`Measure.__copy__(*args)`

`Measure.__delitem__(i)`
Container item deletion with optional time signature adjustment.

`Measure.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Measure.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Measure.__getitem__(i)`
Return component at index i in container. Shallow traversal of container for numeric indices only.

`Measure.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Measure.__iadd__(expr)`
__iadd__ avoids unnecessary copying of structures.

`Measure.__imul__(total)`
Multiply contents of container 'total' times. Return multiplied container.

`Measure.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Measure.__len__()`
Return nonnegative integer number of components in container.

`Measure.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

Measure.**__mul__**(*n*)

Measure.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Measure.**__radd__**(*expr*)
Extend container by contents of expr to the right.

Measure.**__repr__**()
String form of measure with parentheses for interpreter display.

Measure.**__rmul__**(*n*)

Measure.**__setitem__**(*i, expr*)
Container setitem logic with optional time signature adjustment.

Measure setitem logic now adjusts time signatue automatically when `adjust_time_signature_automatically` is true.

Measure.**__str__**()
String form of measure with pipes for single string display.

18.2 Functions

18.2.1 measuretools.all_are_measures

measuretools.**all_are_measures**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad measures:

```
>>> measures = 3 * Measure((3, 4), "c'4 d'4 e'4")
```

```
>>> for measure in measures:
...     measure
...
Measure(3/4, [c'4, d'4, e'4])
Measure(3/4, [c'4, d'4, e'4])
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measuretools.all_are_measures(measures)
True
```

True when *expr* is an empty sequence:

```
>>> measuretools.all_are_measures([])
True
```

Otherwise false:

```
>>> measuretools.all_are_measures('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

18.2.2 measuretools.append_spacer_skip_to_underfull_measure

measuretools.**append_spacer_skip_to_underfull_measure**(*rigid_measure*)
New in version 1.1. Append spacer skip to underfull *measure*:

```
>>> measure = Measure((4, 12), "c'8 d'8 e'8 f'8")
>>> contexttools.detach_time_signature_marks_attached_to_component(measure)
(TimeSignatureMark((4, 12)),)
>>> contexttools.TimeSignatureMark((5, 12))(measure)
TimeSignatureMark((5, 12))(|5/12 c'8 d'8 e'8 f'8|)
>>> measure.is_underfull
True
```

```
>>> measuretools.append_spacer_skip_to_underfull_measure(measure)
Measure(5/12, [c'8, d'8, e'8, f'8, s1 * 1/8])
```

```
>>> f(measure)
{
    \time 5/12
    \scaleDurations #'(2 . 3) {
        c'8
        d'8
        e'8
        f'8
        s1 * 1/8
    }
}
```

Append nothing to nonunderfull *measure*.

Return *measure*.

18.2.3 measuretools.append_spacer_skips_to_underfull_measures_in_expr

measuretools.**append_spacer_skips_to_underfull_measures_in_expr**(*expr*)

New in version 1.1. Append spacer skips to underfull measures in *expr*:

```
>>> staff = Staff(Measure((3, 8), "c'8 d'8 e'8") * 3)
>>> contexttools.detach_time_signature_marks_attached_to_component(staff[1])
(TimeSignatureMark((3, 8)),)
>>> contexttools.TimeSignatureMark((4, 8))(staff[1])
TimeSignatureMark((4, 8))(|4/8 c'8 d'8 e'8|)
>>> contexttools.detach_time_signature_marks_attached_to_component(staff[2])
(TimeSignatureMark((3, 8)),)
>>> contexttools.TimeSignatureMark((5, 8))(staff[2])
TimeSignatureMark((5, 8))(|5/8 c'8 d'8 e'8|)
>>> staff[1].is_underfull
True
>>> staff[2].is_underfull
True
```

```
>>> measuretools.append_spacer_skips_to_underfull_measures_in_expr(staff)
[Measure(4/8, [c'8, d'8, e'8, s1 * 1/8]), Measure(5/8, [c'8, d'8, e'8, s1 * 1/4])]
```

```
>>> f(staff)
\new Staff {
    {
        \time 3/8
        c'8
        d'8
        e'8
    }
    {
        \time 4/8
        c'8
        d'8
        e'8
        s1 * 1/8
    }
    {
        \time 5/8
        c'8
        d'8
        e'8
```

```
            s1 * 1/4
        }
    }
```

Return measures treated.

18.2.4 measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr

measuretools.**apply_full_measure_tuplets_to_contents_of_measures_in_expr**(*expr*, *supplement=None*)

New in version 2.0. Apply full-measure tuplets to contents of measures in *expr*:

```
>>> staff = Staff([Measure((2, 8), "c'8 d'8"), Measure((3, 8), "e'8 f'8 g'8")])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        \time 3/8
        e'8
        f'8
        g'8
    }
}
```

```
>>> measuretools.apply_full_measure_tuplets_to_contents_of_measures_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        {
            c'8
            d'8
        }
    }
    {
        \time 3/8
        {
            e'8
            f'8
            g'8
        }
    }
}
```

Return none.

18.2.5 measuretools.comment_measures_in_container_with_measure_numbers

measuretools.**comment_measures_in_container_with_measure_numbers**(*container*)

New in version 1.1. Comment measures in *container* with measure numbers:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> measuretools.comment_measures_in_container_with_measure_numbers(staff)
```

```
>>> f(staff)
\new Staff {
    % start measure 1
    {
        \time 2/8
        c'8
        d'8
    }
    % stop measure 1
    % start measure 2
    {
        e'8
        f'8
    }
    % stop measure 2
    % start measure 3
    {
        g'8
        a'8
    }
    % stop measure 3
}
```

Return none.

18.2.6 measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets

`measuretools.`**`extend_measures_in_expr_and_apply_full_measure_tuplets`**(*expr*, *supplement*)

New in version 2.0. Extend measures in *expr* with *supplement* and apply full-measure tuplets to contents of measures:

```
>>> staff = Staff([Measure((2, 8), "c'8 d'8"), Measure((3, 8), "e'8 f'8 g'8")])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        \time 3/8
        e'8
        f'8
        g'8
    }
}
```

```
>>> supplement = [Rest((1, 16))]
>>> measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets(
... staff, supplement)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        \times 4/5 {
            c'8
            d'8
            r16
        }
    }
    {
        \time 3/8
        \fraction \times 6/7 {
```

```
                e'8
                f'8
                g'8
                r16
            }
        }
    }
```

Return none.

18.2.7 measuretools.fill_measures_in_expr_with_full_measure_spacer_skips

measuretools.**fill_measures_in_expr_with_full_measure_spacer_skips**(*expr*, *iterctrl=None*)

New in version 1.1. Fill measures in *expr* with full-measure spacer skips.

18.2.8 measuretools.fill_measures_in_expr_with_minimal_number_of_notes

measuretools.**fill_measures_in_expr_with_minimal_number_of_notes**(*expr*, *decrease_durations_monotonically=T*, *iterctrl=None*)

New in version 1.1. Fill measures in *expr* with minimal number of notes that decrease durations monotonically:

```
>>> measure = Measure((5, 18), [])
```

```
>>> measuretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=True)
```

```
>>> f(measure)
{
        ime 5/18
    \scaleDurations #'(8 . 9) {
        c'4 ~
        c'16
    }
}
```

Fill measures in *expr* with minimal number of notes that increase durations monotonically:

```
>>> measure = Measure((5, 18), [])
```

```
>>> measuretools.fill_measures_in_expr_with_minimal_number_of_notes(
...     measure, decrease_durations_monotonically=False)
```

```
>>> f(measure)
{
        ime 5/18
    \scaleDurations #'(8 . 9) {
        c'16 ~
        c'4
    }
}
```

Return none.

18.2.9 measuretools.fill_measures_in_expr_with_repeated_notes

measuretools.**fill_measures_in_expr_with_repeated_notes**(*expr*, *written_duration*, *iterctrl=None*)

New in version 1.1. Fill measures in *expr* with repeated notes.

18.2.10 measuretools.fill_measures_in_expr_with_time_signature_denominator_notes

measuretools.**fill_measures_in_expr_with_time_signature_denominator_notes**(*expr*, *itercctrl=None*)

New in version 1.1. Fill measures in *expr* with time signature denominator notes:

```
>>> staff = Staff([Measure((3, 4), []), Measure((3, 16), []), Measure((3, 8), [])])
>>> measuretools.fill_measures_in_expr_with_time_signature_denominator_notes(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 3/4
        c'4
        c'4
        c'4
    }
    {
        \time 3/16
        c'16
        c'16
        c'16
    }
    {
        \time 3/8
        c'8
        c'8
        c'8
    }
}
```

Delete existing contents of measures in *expr*.

Return none.

18.2.11 measuretools.fuse_contiguous_measures_in_container_cyclically_by_counts

measuretools.**fuse_contiguous_measures_in_container_cyclically_by_counts**(*container*, *counts*, *mark=False*)

New in version 1.1. Fuse contiguous measures in *container* cyclically by *counts*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 5)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
    {
        b'8
        c''8
    }
    {
        d''8
```

```
            e''8
        }
    }
```

```
>>> counts = (2, 1)
>>> measuretools.fuse_contiguous_measures_in_container_cyclically_by_counts(staff, counts)
```

```
>>> f(staff)
\new Staff {
    {
        \time 4/8
        c'8
        d'8
        e'8
        f'8
    }
    {
        \time 2/8
        g'8
        a'8
    }
    {
        \time 4/8
        b'8
        c''8
        d''8
        e''8
    }
}
```

Return none.

Set *mark* to true to mark fused measures for later reference.

18.2.12 measuretools.fuse_measures

measuretools.**fuse_measures**(*measures*)

New in version 1.1. Fuse *measures*:

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (2, 16)]))
>>> measuretools.fill_measures_in_expr_with_repeated_notes(staff, Duration(1, 16))
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> beamtools.BeamSpanner(staff.leaves)
BeamSpanner(c'16, d'16, e'16, f'16)
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/8
        c'16 [
        d'16
    }
    {
        \time 2/16
        e'16
        f'16 ]
    }
}
```

```
>>> measuretools.fuse_measures(staff[:])
Measure(2/8, [c'16, d'16, e'16, f'16])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'16 [
        d'16
```

```
            e'16
            f'16 ]
        }
    }
```

Return new measure.

Allow parent-contiguous *measures*.

Allow outside-of-score *measures*.

Do not define measure fusion across intervening container boundaries.

Calculate best new time signature.

Instantiate new measure.

Give *measures* contents to new measure.

Give *measures* dominant spanners to new measure.

Give *measures* parentage to new measure.

Leave *measures* empty, unspanned and outside-of-score.

18.2.13 measuretools.get_first_measure_in_improper_parentage_of_component

measuretools.**get_first_measure_in_improper_parentage_of_component**(*component*)
New in version 2.0. Get first measure in improper parentage of *component*:

```
>>> measure = Measure((2, 4), "c'8 d'8 e'8 f'8")
>>> staff = Staff([measure])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'8
        d'8
        e'8
        f'8
    }
}
```

```
>>> measuretools.get_first_measure_in_improper_parentage_of_component(staff.leaves[0])
Measure(2/4, [c'8, d'8, e'8, f'8])
```

Return measure or none.

18.2.14 measuretools.get_first_measure_in_proper_parentage_of_component

measuretools.**get_first_measure_in_proper_parentage_of_component**(*component*)
New in version 2.0. Get first measure in proper parentage of *component*:

```
>>> measure = Measure((2, 4), "c'8 d'8 e'8 f'8")
>>> staff = Staff([measure])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'8
        d'8
        e'8
        f'8
    }
}
```

```
>>> measuretools.get_first_measure_in_proper_parentage_of_component(staff.leaves[0])
Measure(2/4, [c'8, d'8, e'8, f'8])
```

Return measure or none.

18.2.15 measuretools.get_likely_multiplier_of_components

measuretools.**get_likely_multiplier_of_components**(*components*)

New in version 2.0. Get likely multiplier of *components*:

```
>>> staff = Staff("c'8.. d'8.. e'8.. f'8..")
```

```
>>> f(staff)
\new Staff {
    c'8..
    d'8..
    e'8..
    f'8..
}
```

```
>>> measuretools.get_likely_multiplier_of_components(staff[:])
Multiplier(7, 4)
```

Return 1 when no multiplier is likely:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> measuretools.get_likely_multiplier_of_components(staff[:])
Multiplier(1, 1)
```

Return none when more than one multiplier is likely:

```
>>> staff = Staff(notetools.make_notes([0, 2, 4, 5], [(3, 16), (7, 32)]))
```

```
>>> f(staff)
\new Staff {
    c'8.
    d'8..
    e'8.
    f'8..
}
```

```
>>> measuretools.get_likely_multiplier_of_components(staff[:]) is None
True
```

Return multiplier or none.

18.2.16 measuretools.get_measure_that_starts_with_container

measuretools.**get_measure_that_starts_with_container**(*container*)

New in version 2.11. Get measure that starts with *container*.

Return measure or none.

18.2.17 measuretools.get_measure_that_stops_with_container

measuretools.**get_measure_that_stops_with_container**(*container*)

New in version 2.11. Get measure that stops with *container*.

Return measure or none.

18.2.18 measuretools.get_next_measure_from_component

measuretools.**get_next_measure_from_component**(*component*)

New in version 1.1. Get next measure from *component*.

When *component* is a voice, staff or other sequential context, and when *component* contains a measure, return first measure in *component*. This starts the process of forwards measure iteration.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_next_measure_from_component(staff)
Measure(2/8, [c'8, d'8])
```

When *component* is voice, staff or other sequential context, and when *component* contains no measure, raise missing measure error.

When *component* is a measure and there is a measure immediately following *component*, return measure immediately following component.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff[0]) is None
True
```

When *component* is a measure and there is no measure immediately following *component*, return `None`.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff[-1])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is a measure in the parentage of *component*, return the measure in the parentage of *component*.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff.leaves[0])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is no measure in the parentage of *component*, raise missing measure error.

18.2.19 measuretools.get_nth_measure_in_expr

measuretools.**get_nth_measure_in_expr**(*expr*, *n=0*)

New in version 2.0. Get nth measure in *expr*:

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
```

```
            e'8
            f'8
        }
        {
            g'8
            a'8
        }
    }
```

Read forward for positive values of *n*.

```
>>> for n in range(3):
...     measuretools.get_nth_measure_in_expr(staff, n)
...
Measure(2/8, [c'8, d'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [g'8, a'8])
```

Read backward for negative values of *n*.

```
>>> for n in range(3, -1, -1):
...     measuretools.get_nth_measure_in_expr(staff, n)
...
Measure(2/8, [g'8, a'8])
Measure(2/8, [e'8, f'8])
Measure(2/8, [c'8, d'8])
```

Return measure.

18.2.20 measuretools.get_one_indexed_measure_number_in_expr

measuretools.**get_one_indexed_measure_number_in_expr**(*expr*, *measure_number*)

New in version 2.0. Get one-indexed *measure_number* in *expr*:

```
>>> t = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
```

```
>>> f(t)
\new Staff {
    {
        \time 2/8
        c'8
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8
    }
}
>>> measuretools.get_one_indexed_measure_number_in_expr(t, 3)
Measure(2/8, [g'8, a'8])
```

Note that measures number from `1`.

18.2.21 measuretools.get_previous_measure_from_component

measuretools.**get_previous_measure_from_component**(*component*)

New in version 1.1. Get previous measure from *component*.

When *component* is voice, staff or other sequential context, and when *component* contains a measure, return last measure in *component*. This starts the process of backwards measure iteration.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff)
Measure(2/8, [e'8, f'8])
```

When *component* is voice, staff or other sequential context, and when *component* contains no measure, raise missing measure error.

When *component* is a measure and there is a measure immediately preceeding *component*, return measure immediately preceeding component.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff[-1])
Measure(2/8, [c'8, d'8])
```

When *component* is a measure and there is no measure immediately preceeding *component*, return `None`.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff[0]) is None
True
```

When *component* is a leaf and there is a measure in the parentage of *component*, return the measure in the parentage of *component*.

```
>>> staff = Staff(Measure((2, 8), notetools.make_repeated_notes(2)) * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
>>> measuretools.get_previous_measure_from_component(staff.leaves[0])
Measure(2/8, [c'8, d'8])
```

When *component* is a leaf and there is no measure in the parentage of *component*, raise missing measure error.

18.2.22 measuretools.list_time_signatures_of_measures_in_expr

`measuretools.`**`list_time_signatures_of_measures_in_expr`**(*components*)

New in version 2.0. List time signatures of measures in *expr*:

```
>>> staff = Staff('abj: | 2/8 c8 d8 || 3/8 c8 d8 e8 || 4/8 c8 d8 e8 f8 |')
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c8
        d8
    }
    {
        \time 3/8
        c8
        d8
        e8
    }
    {
        \time 4/8
        c8
        d8
        e8
        f8
    }
}
```

```
>>> for x in measuretools.list_time_signatures_of_measures_in_expr(staff): x
...
TimeSignatureMark((2, 8))(|2/8 c8 d8|)
TimeSignatureMark((3, 8))(|3/8 c8 d8 e8|)
TimeSignatureMark((4, 8))(|4/8 c8 d8 e8 f8|)
```

Return list of zero or more time signatures.

18.2.23 measuretools.make_measures_with_full_measure_spacer_skips

measuretools.**make_measures_with_full_measure_spacer_skips**(*time_signatures*)

New in version 1.1. Make measures with full-measure spacer skips from *time_signatures*:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
```

```
>>> staff = Staff(measures)
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

Return list of rigid measures.

18.2.24 measuretools.measure_to_one_line_input_string

measuretools.**measure_to_one_line_input_string**(*measure*)

New in version 2.6. Change *measure* to one-line input string:

```
>>> measure = Measure((4, 4), "c'4 d'4 e'4 f'4")
```

```
>>> input_string = measuretools.measure_to_one_line_input_string(measure)
```

```
>>> print input_string
Measure((4, 4), "c'4 d'4 e'4 f'4")
```

```
>>> new_measure = eval(input_string)
```

```
>>> new_measure
Measure(4/4, [c'4, d'4, e'4, f'4])
```

```
>>> f(new_measure)
{
    \time 4/4
    c'4
    d'4
    e'4
    f'4
}
```

The purpose of this function is to create an evaluable string from simple measures.

Spanners, articulations and overrides not supported.

Return string.

18.2.25 measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature

measuretools.**move_full_measure_tuplet_prolation_to_measure_time_signature**(*expr*)

New in version 1.1. Move prolation of full-measure tuplet to time signature of measure.

Measures usually become non-power-of-two as as result:

```
>>> t = Measure((2, 8), [tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")])
>>> measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature(t)
```

```
>>> f(t)
{
    \time 3/12
    \scaleDurations #'(2 . 3) {
        c'8
        d'8
        e'8
    }
}
```

Return none.

18.2.26 measuretools.move_measure_prolation_to_full_measure_tuplet

measuretools.**move_measure_prolation_to_full_measure_tuplet**(*expr*)

New in version 2.0. Move measure prolation to full-measure tuplet.

Turn non-power-of-two measures into power-of-two measures containing a single fixed-duration tuplet.

Note that not all non-power-of-two measures can be made power-of-two.

Returns None because processes potentially many measures.

18.2.27 measuretools.multiply_and_scale_contents_of_measures_in_expr

measuretools.**multiply_and_scale_contents_of_measures_in_expr**(*expr*, *multiplier_pairs*)

New in version 1.1. Multiply and scale contents of measures in *expr* by *multiplier_pairs*.

The *multiplier_pairs* argument must be a list of `(contents_multiplier, denominator_multiplier)` pairs.

Both *contents_multiplier* and *denominator_multiplier* must be positive integers.

Example 1. Multiply measure contents by `3`. Scale time signature denominator by `3`:

```
>>> measure = Measure((3, 16), "c'16 c'16 c'16")
```

```
>>> measuretools.multiply_and_scale_contents_of_measures_in_expr(measure, [(3, 3)])
[Measure(9/48, [c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32])]
```

Example 2. Multiply measure contents by `3`. Scale time signature denominator by `2`:

```
>>> measure = Measure((3, 16), "c'16 c'16 c'16")
```

```
>>> measuretools.multiply_and_scale_contents_of_measures_in_expr(measure, [(3, 2)])
[Measure(9/32, [c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32])]
```

Example 3. Multiply measure contents `3`:

```
>>> measure = Measure((3, 16), "c'16 c'16 c'16")
```

```
>>> measuretools.multiply_and_scale_contents_of_measures_in_expr(measure, [(3, 1)])
[Measure(9/16, [c'16, c'16, c'16, c'16, c'16, c'16, c'16, c'16, c'16])]
```

Return list of measures changed.

18.2.28 measuretools.multiply_contents_of_measures_in_expr

measuretools.**multiply_contents_of_measures_in_expr**(*expr*, *n*)

New in version 1.1. Multiply contents `n - 1` times and adjust time signature of every measure in *expr*:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> beamtools.BeamSpanner(measure.leaves)
BeamSpanner(c'8, d'8, e'8)
```

```
>>> f(measure)
{
    \time 3/8
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> measuretools.multiply_contents_of_measures_in_expr(measure, 3)
```

```
>>> f(measure)
{
    \time 9/8
    c'8 [
    d'8
    e'8 ]
    c'8 [
    d'8
    e'8 ]
    c'8 [
    d'8
    e'8 ]
}
```

Return none.

18.2.29 measuretools.pad_measures_in_expr_with_rests

measuretools.**pad_measures_in_expr_with_rests**(*expr*, *front*, *back*, *splice=False*)

New in version 1.1. Pad measures in *expr* with rests.

Iterate all measures in *expr*. Insert rest with duration equal to *front* at beginning of each measure. Insert rest with duation aqual to *back* at end of each measure.

Set *front* to a positive rational or none. Set *back* to a positive rational or none.

Note that this function is designed to help create regularly spaced charts and tables of musical materials. This function makes most sense when used on anonymous measures or dynamic measures.

```
>>> t = Staff(2 * Measure((2, 8), "c'8 d'8"))
>>> front, back = Duration(1, 32), Duration(1, 64)
>>> measuretools.pad_measures_in_expr_with_rests(t, front, back)
```

```
>>> f(t)
\new Staff {
    {
        \time 19/64
        r32
        c'8
        d'8
        r64
    }
    {
        r32
        c'8
        d'8
        r64
    }
}
```

Works when measures contain stacked voices:

```
>>> measure = Measure((2, 8), 2 * Voice(notetools.make_repeated_notes(2)))
>>> measure.is_parallel = True
>>> t = Staff(measure * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
>>> measuretools.pad_measures_in_expr_with_rests(
...     t, Duration(1, 32), Duration(1, 64))
```

```
>>> f(t)
\new Staff {
    <<
        \time 19/64
        \new Voice {
            r32
            c'8
            d'8
            r64
        }
        \new Voice {
            r32
            e'8
            f'8
            r64
        }
    >>
    <<
        \new Voice {
            r32
            g'8
            a'8
            r64
        }
        \new Voice {
            r32
            b'8
            c''8
            r64
        }
    >>
}
```

Set the optional *splice* keyword to `True` to extend edge spanners over newly inserted rests:

```
>>> t = Measure((2, 8), "c'8 d'8")
>>> beamtools.BeamSpanner(t[:])
BeamSpanner(c'8, d'8)
>>> measuretools.pad_measures_in_expr_with_rests(
...     t, Duration(1, 32), Duration(1, 64), splice=True)
```

```
>>> f(t)
{
    \time 19/64
    r32 [
    c'8
    d'8
    r64 ]
}
```

Return none.

Raise value when *front* is neither a positive rational nor none.

Raise value when *back* is neither a positive rational nor none.

18.2.30 measuretools.pad_measures_in_expr_with_skips

`measuretools.`**`pad_measures_in_expr_with_skips`**(*expr*, *front*, *back*, *splice=False*)

New in version 2.0. Pad measures in *expr* with skips.

Iterate all measures in *expr*. Insert skip with duration equal to *front* at beginning of each measure. Insert skip with duation aqual to *back* at end of each measure.

Set *front* to a positive rational or none. Set *back* to a positive rational or none.

Note that this function is designed to help create regularly spaced charts and tables of musical materials. This function makes most sense when used on anonymous measures and dynamic measures.

```
>>> t = Staff(2 * Measure((2, 8), "c'8 d'8"))
>>> front, back = Duration(1, 32), Duration(1, 64)
>>> measuretools.pad_measures_in_expr_with_skips(t, front, back)
```

```
>>> f(t)
\new Staff {
    {
        \time 19/64
        s32
        c'8
        d'8
        s64
    }
    {
        s32
        c'8
        d'8
        s64
    }
}
```

Works when measures contain stacked voices:

```
>>> measure = Measure((2, 8), 2 * Voice(notetools.make_repeated_notes(2)))
>>> measure.is_parallel = True
>>> t = Staff(measure * 2)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
>>> measuretools.pad_measures_in_expr_with_skips(
...     t, Duration(1, 32), Duration(1, 64))
```

```
>>> f(t)
\new Staff {
    <<
        \time 19/64
        \new Voice {
            s32
            c'8
            d'8
            s64
        }
        \new Voice {
            s32
            e'8
            f'8
            s64
        }
    >>
    <<
        \new Voice {
            s32
            g'8
            a'8
            s64
        }
        \new Voice {
            s32
            b'8
            c''8
            s64
        }
    >>
}
```

Set the optional *splice* keyword to `True` to extend edge spanners over newly inserted skips:

```
>>> t = Measure((2, 8), "c'8 d'8")
>>> beamtools.BeamSpanner(t[:])
BeamSpanner(c'8, d'8)
>>> measuretools.pad_measures_in_expr_with_skips(
...     t, Duration(1, 32), Duration(1, 64), splice = True)
```

```
>>> f(t)
{
    \time 19/64
    s32 [
    c'8
    d'8
    s64 ]
}
```

Return none.

Raise value error when *front* is neither a positive rational nor none.

Raise value error when *back* is neither a positive rational nor none.

18.2.31 measuretools.replace_contents_of_measures_in_expr

measuretools.**replace_contents_of_measures_in_expr**(*expr*, *new_contents*)
New in version 1.1. Replace contents of measures in *expr* with *new_contents*:

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (3, 16)]))
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 3/16
        s1 * 3/16
    }
}
```

```
>>> notes = [Note("c'16"), Note("d'16"), Note("e'16"), Note("f'16")]
>>> measuretools.replace_contents_of_measures_in_expr(staff, notes)
[Measure(1/8, [c'16, d'16]), Measure(3/16, [e'16, f'16, s1 * 1/16])]
```

```
>>> f(staff)
\new Staff {
    {
        \time 1/8
        c'16
        d'16
    }
    {
        \time 3/16
        e'16
        f'16
        s1 * 1/16
    }
}
```

Preserve duration of all measures.

Skip measures that are too small.

Pad extra space at end of measures with spacer skip.

If not enough measures raise stop iteration.

Return measures iterated.

18.2.32 measuretools.report_time_signature_distribution

`measuretools.report_time_signature_distribution(expr)`

New in version 2.0. Report time signature distribution of *expr*:

```
>>> staff = Staff(r"abj: | 2/4 c'4 d'4 || 2/4 e'4 f'4 || 2/4 g'2 |" \
...     "| 5/8 c'8 d'8 e'8 f'8 g'8 |")
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'4
        d'4
    }
    {
        e'4
        f'4
    }
    {
        g'2
    }
    {
        \time 5/8
        c'8
        d'8
        e'8
        f'8
        g'8
    }
}
```

```
>>> print measuretools.report_time_signature_distribution(staff)
    2/4 3
    5/8 1
```

Return string.

18.2.33 measuretools.scale_contents_of_measures_in_expr

`measuretools.scale_contents_of_measures_in_expr(expr, multiplier=1)`

New in version 2.0. Scale contents of measures in *expr* by *multiplier*.

Iterate expr. For every measure in expr first multiply the measure time siganture by *multiplier* and then scale measure contents to fit the new time signature.

Extend `containertools.scale_contents_of_container()`.

Return none.

18.2.34 measuretools.scale_measure_and_adjust_time_signature

`measuretools.scale_measure_and_adjust_time_signature(measure, multiplier=1)`

New in version 2.0. Scale *measure* by *multiplier* and adjust time signature:

```
>>> t = Measure((3, 8), "c'8 d'8 e'8")
>>> measuretools.scale_measure_and_adjust_time_signature(t, Duration(2, 3))
Measure(3/12, [c'8, d'8, e'8])
```

```
>>> f(t)
{
\time 3/12
\scaleDurations #'(2 . 3) {
    c'8
    d'8
    e'8
```

```
    }
}
```

Return *measure*.

18.2.35 measuretools.scale_measure_denominator_and_adjust_measure_contents

measuretools.**scale_measure_denominator_and_adjust_measure_contents**(*measure*, *factor*)

New in version 1.1. Change power-of-two *measure* to non-power-of-two measure with new denominator *factor*:

```
>>> measure = Measure((2, 8), "c'8 d'8")
>>> beamtools.BeamSpanner(measure.leaves)
BeamSpanner(c'8, d'8)
```

```
>>> f(measure)
{
    \time 2/8
    c'8 [
    d'8 ]
}
```

```
>>> measuretools.scale_measure_denominator_and_adjust_measure_contents(measure, 3)
Measure(3/12, [c'8., d'8.])
```

```
>>> f(measure)
{
    \time 3/12
    \scaleDurations #'(2 . 3) {
        c'8. [
        d'8. ]
    }
}
```

Treat new denominator *factor* like clever form of `1`: `3/3` or `5/5` or `7/7`, etc.

Preserve *measure* prolated duration.

Derive new *measure* multiplier.

Scale *measure* contents.

Pick best new time signature.

18.2.36 measuretools.set_always_format_time_signature_of_measures_in_expr

measuretools.**set_always_format_time_signature_of_measures_in_expr**(*expr*, *value=True*)

New in version 2.9. Set *always_format_time_signature* of measures in *expr* to boolean *value*.

Return none.

18.2.37 measuretools.set_measure_denominator_and_adjust_numerator

measuretools.**set_measure_denominator_and_adjust_numerator**(*measure*, *denominator*)

New in version 1.1. Set *measure* time signature *denominator* and multiply time signature numerator accordingly:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
>>> beamtools.BeamSpanner(measure.leaves)
BeamSpanner(c'8, d'8, e'8)
```

```
>>> f(measure)
{
    \time 3/8
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> measuretools.set_measure_denominator_and_adjust_numerator(measure, 16)
Measure(6/16, [c'8, d'8, e'8])
```

```
>>> f(measure)
{
    \time 6/16
    c'8 [
    d'8
    e'8 ]
}
```

Leave *measure* contents unchanged.

Return *measure*.

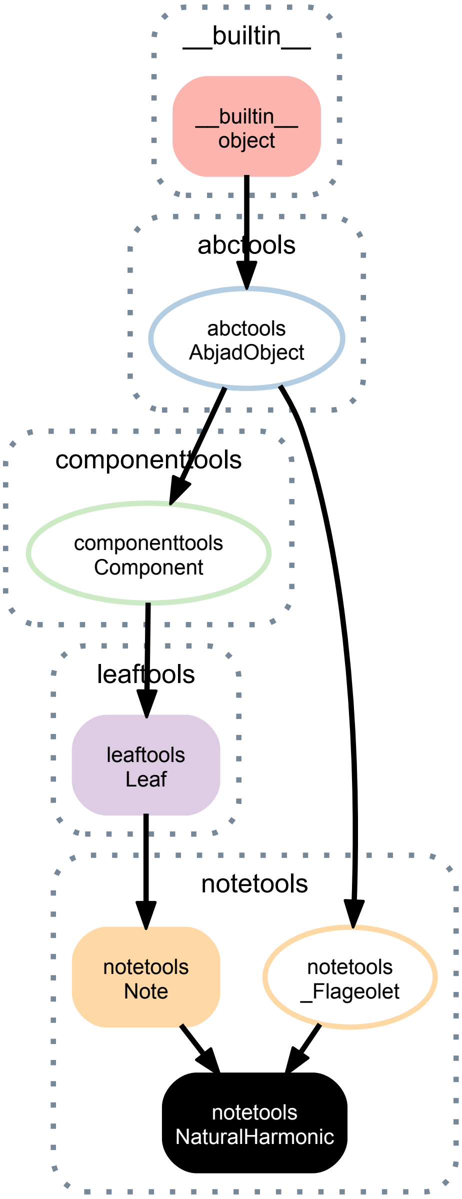
CHAPTER

NINETEEN

NOTETOOLS

19.1 Concrete Classes

19.1.1 notetools.NaturalHarmonic



class notetools.**NaturalHarmonic**(**args*)

Abjad model of natural harmonic.

Initialize natural harmonic by hand:

```
>>> notetools.NaturalHarmonic("cs'8.")
NaturalHarmonic(cs', 8.)
```

Initialize natural harmonic from note:

```
>>> note = Note("cs'8.")
```

```
>>> notetools.NaturalHarmonic(note)
NaturalHarmonic(cs', 8.)
```

Natural harmonics are immutable.

Read-only properties

NaturalHarmonic.**descendants**

Read-only reference to component descendants score selection.

NaturalHarmonic.**duration**

NaturalHarmonic.**duration_in_seconds**

NaturalHarmonic.**fingered_pitch**

Read-only fingered pitch of note:

```
>>> staff = Staff("d''8 e''8 f''8 g''8")
>>> piccolo = instrumenttools.Piccolo()(staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    d'8
    e'8
    f'8
    g'8
}
```

```
>>> staff[0].fingered_pitch
NamedChromaticPitch("d'")
```

Return named chromatic pitch.

NaturalHarmonic.**leaf_index**

NaturalHarmonic.**lilypond_format**

NaturalHarmonic.**lineage**

Read-only reference to component lineage score selection.

NaturalHarmonic.**multiplied_duration**

NaturalHarmonic.**override**

Read-only reference to LilyPond grob override component plug-in.

NaturalHarmonic.**parent**

NaturalHarmonic.**parentage**

Read-only reference to component parentage score selection.

NaturalHarmonic.**preprolated_duration**

NaturalHarmonic.**prolation**

NaturalHarmonic.**set**
Read-only reference LilyPond context setting component plug-in.

NaturalHarmonic.**spanners**
Read-only reference to unordered set of spanners attached to component.

NaturalHarmonic.**storage_format**
Storage format of Abjad object.

Return string.

NaturalHarmonic.**suono_reale**
Actual sound of the harmonic when played.

NaturalHarmonic.**timespan**
Read-only timespan of component.

NaturalHarmonic.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

NaturalHarmonic.**duration_multiplier**

NaturalHarmonic.**note_head**
Get note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head
NoteHead("cs''")
```

Set note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head = 14
>>> note
Note("d''8.")
```

NaturalHarmonic.**sounding_pitch**
Get sounding pitch of note:

```
>>> staff = Staff("d''8 e''8 f''8 g''8")
>>> piccolo = instrumenttools.Piccolo()(staff)
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    d'8
    e'8
    f'8
    g'8
}
>>> staff[0].sounding_pitch
NamedChromaticPitch("d''")
```

Set sounding pitch of note:

```
>>> staff[0].sounding_pitch = "dqs''"
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    dqs'8
    e'8
    f'8
```

```
        g'8
    }
```

NaturalHarmonic.**written_duration**

NaturalHarmonic.**written_pitch**

Get named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch
NamedChromaticPitch("cs''")
```

Set named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch = 14
>>> note
Note("d''8.")
```

NaturalHarmonic.**written_pitch_indication_is_at_sounding_pitch**

NaturalHarmonic.**written_pitch_indication_is_nonsemantic**

Special methods

NaturalHarmonic.**__and__**(*arg*)

NaturalHarmonic.**__copy__**(**args*)

NaturalHarmonic.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

NaturalHarmonic.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

NaturalHarmonic.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

NaturalHarmonic.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

NaturalHarmonic.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

NaturalHarmonic.**__mul__**(*n*)

NaturalHarmonic.**__ne__**(*expr*)

Defined equal to the opposite of equality.

Return boolean.

NaturalHarmonic.**__or__**(*arg*)

NaturalHarmonic.**__repr__**()

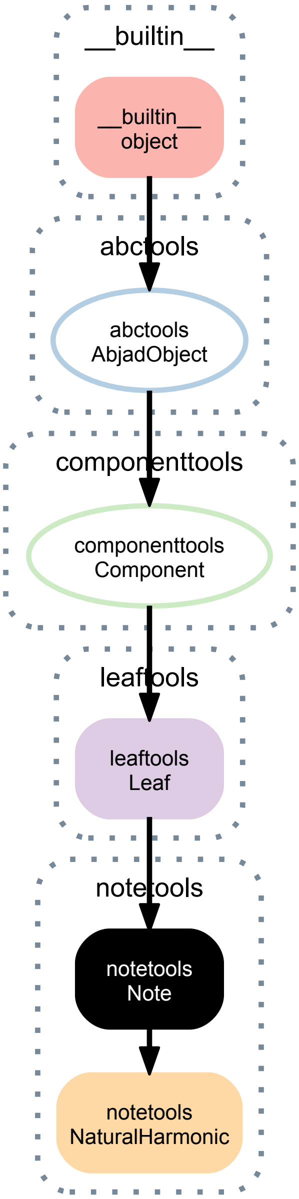
NaturalHarmonic.**__rmul__**(*n*)

NaturalHarmonic.**__str__**()

NaturalHarmonic.**__sub__**(*arg*)

NaturalHarmonic.**__xor__**(*arg*)

19.1.2 notetools.Note



class notetools.**Note**(**args*, ***kwargs*)

New in version 1.0. Abjad model of a note:

```
>>> note = Note("cs''8.")
```

```
>>> note
Note("cs''8.")
```

```
>>> show(note)
```

Return Note instance.

Read-only properties

Note.**descendants**
Read-only reference to component descendants score selection.

Note.**duration**

Note.**duration_in_seconds**

Note.**fingered_pitch**
Read-only fingered pitch of note:

```
>>> staff = Staff("d''8 e''8 f''8 g''8")
>>> piccolo = instrumenttools.Piccolo()(staff)
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    d'8
    e'8
    f'8
    g'8
}
```

```
>>> staff[0].fingered_pitch
NamedChromaticPitch("d'")
```

Return named chromatic pitch.

Note.**leaf_index**

Note.**lilypond_format**

Note.**lineage**
Read-only reference to component lineage score selection.

Note.**multiplied_duration**

Note.**override**
Read-only reference to LilyPond grob override component plug-in.

Note.**parent**

Note.**parentage**
Read-only reference to component parentage score selection.

Note.**preprolated_duration**

Note.**prolation**

Note.**set**
Read-only reference LilyPond context setting component plug-in.

Note.**spanners**
Read-only reference to unordered set of spanners attached to component.

Note.**storage_format**
Storage format of Abjad object.

Return string.

Note.**timespan**
Read-only timespan of component.

Note.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

Note.**duration_multiplier**

Note.**note_head**

Get note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head
NoteHead("cs''")
```

Set note head of note:

```
>>> note = Note(13, (3, 16))
>>> note.note_head = 14
>>> note
Note("d''8.")
```

Note.**sounding_pitch**

Get sounding pitch of note:

```
>>> staff = Staff("d''8 e''8 f''8 g''8")
>>> piccolo = instrumenttools.Piccolo()(staff)
```

```
>>> instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch(staff)
```

```
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    d'8
    e'8
    f'8
    g'8
}
>>> staff[0].sounding_pitch
NamedChromaticPitch("d''")
```

Set sounding pitch of note:

```
>>> staff[0].sounding_pitch = "dqs''"
>>> f(staff)
\new Staff {
    \set Staff.instrumentName = \markup { Piccolo }
    \set Staff.shortInstrumentName = \markup { Picc. }
    dqs'8
    e'8
    f'8
    g'8
}
```

Note.**written_duration**

Note.**written_pitch**

Get named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch
NamedChromaticPitch("cs''")
```

Set named pitch of note:

```
>>> note = Note(13, (3, 16))
>>> note.written_pitch = 14
>>> note
Note("d''8.")
```

Note.**written_pitch_indication_is_at_sounding_pitch**

Note.**written_pitch_indication_is_nonsemantic**

Special methods

Note.**__and__**(*arg*)

Note.**__copy__**(**args*)

Note.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Note.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Note.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Note.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Note.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Note.**__mul__**(*n*)

Note.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Note.**__or__**(*arg*)

Note.**__repr__**()

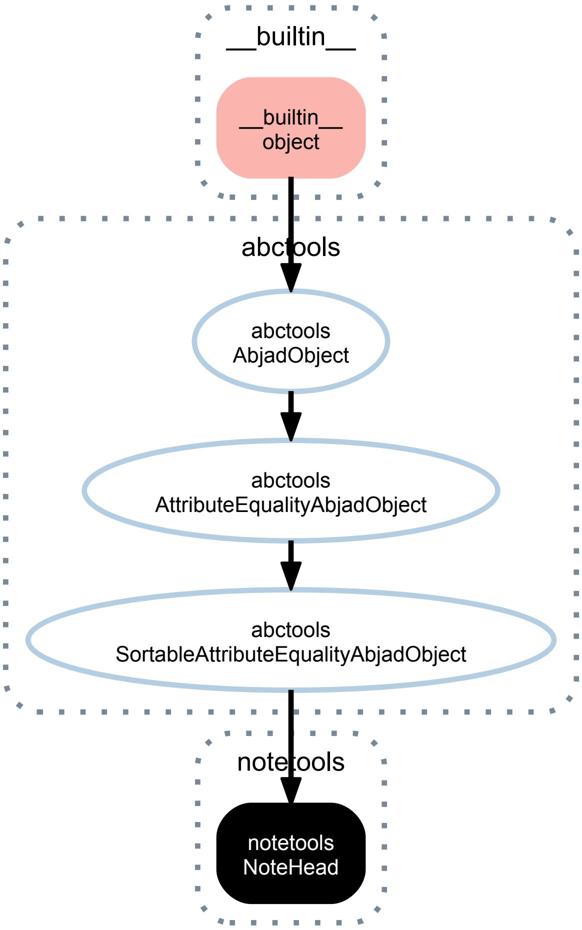
Note.**__rmul__**(*n*)

Note.**__str__**()

Note.**__sub__**(*arg*)

Note.**__xor__**(*arg*)

19.1.3 notetools.NoteHead



class `notetools.`**`NoteHead`**(*written_pitch=None*, *client=None*, *is_cautionary=False*, *is_forced=False*, *tweak_pairs=()*)

Abjad model of a note head:

```
>>> notetools.NoteHead(13)
NoteHead("cs''")
```

Note heads are immutable.

Read-only properties

`NoteHead.`**`lilypond_format`**

Read-only LilyPond input format of note head:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.lilypond_format
"cs''"
```

Return string.

`NoteHead.`**`named_chromatic_pitch`**

Read-only named chromatic pitch equal to note head:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.named_chromatic_pitch
NamedChromaticPitch("cs''")
```

Return named chromatic pitch.

NoteHead.**storage_format**
Storage format of Abjad object.

Return string.

NoteHead.**tweak**
Read-only LilyPond tweak reservoir:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.tweak
LilyPondTweakReservoir()
```

Return LilyPond tweak reservoir.

Read/write properties

NoteHead.**is_cautionary**
Get cautionary accidental flag:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.is_cautionary
False
```

Set cautionary accidental flag:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.is_cautionary = True
```

Return boolean.

NoteHead.**is_forced**

NoteHead.**written_pitch**
Get named pitch of note head:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.written_pitch
NamedChromaticPitch("cs''")
```

Set named pitch of note head:

```
>>> note_head = notetools.NoteHead("cs''")
>>> note_head.written_pitch = "d''"
>>> note_head.written_pitch
NamedChromaticPitch("d''")
```

Set pitch token.

Special methods

NoteHead.**__copy__**(**args*)

NoteHead.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NoteHead.**__repr__**()

NoteHead.**__str__**()

19.2 Functions

19.2.1 notetools.add_artificial_harmonic_to_note

notetools.**add_artificial_harmonic_to_note**(*note*, *melodic_diatonic_interval=None*)
Add artifical harmonic to *note* at *melodic_diatonic_interval*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> notetools.add_artificial_harmonic_to_note(staff[0])
Chord("<c' f'>8")
```

```
>>> f(staff)
\new Staff {
    <
        c'
        \tweak #'style #'harmonic
        f'
    >8 [
    d'8
    e'8
    f'8 ]
}
```

When `melodic_diatonic_interval=None` set to a perfect fourth.

Create new artificial harmonic chord from *note*.

Move parentage and spanners from *note* to artificial harmonic chord.

Return artificial harmonic chord.

19.2.2 notetools.all_are_notes

notetools.**all_are_notes**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad notes:

```
>>> notes = [Note("c'4"), Note("d'4"), Note("e'4")]
```

```
>>> notetools.all_are_notes(notes)
True
```

True when *expr* is an empty sequence:

```
>>> notetools.all_are_notes([])
True
```

Otherwise false:

```
>>> notetools.all_are_notes('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

19.2.3 notetools.make_accelerating_notes_with_lilypond_multipliers

`notetools.`**`make_accelerating_notes_with_lilypond_multipliers`**(*pitches*, *total*, *start*, *stop*, *exp='cosine'*, *written=None*)

Make accelerating notes with LilyPond multipliers:

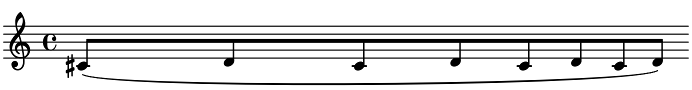
```
>>> pitches = ['C#4', 'D4']
>>> total = Duration(4, 4)
>>> start = Duration(1, 4)
>>> stop = Duration(1, 16)
>>> args = [pitches, total, start, stop]
```

```
>>> notes = notetools.make_accelerating_notes_with_lilypond_multipliers(*args)
```

```
>>> staff = Staff(notes)
>>> beam = beamtools.BeamSpanner(staff[:])
>>> slur = spannertools.SlurSpanner(staff[:])
```

```
>>> f(staff)
\new Staff {
    cs'8 * 245/128 [ (
    d'8 * 109/64
    cs'8 * 161/128
    d'8 * 115/128
    cs'8 * 87/128
    d'8 * 9/16
    cs'8 * 1/2
    d'8 * 61/128 ] )
}
```

```
>>> show(staff)
```



Set note pitches cyclically from *pitches*.

Return as many interpolation values as necessary to fill the *total* duration requested.

Interpolate durations from *start* to *stop*.

Set note durations to *written* duration times computed interpolated multipliers.

Interprete *written=None* as eighth notes.

Return list of notes.

19.2.4 notetools.make_notes

`notetools.`**`make_notes`**(*pitches*, *durations*, *decrease_durations_monotonically=True*)

Make notes according to *pitches* and *durations*.

Cycle through *pitches* when the length of *pitches* is less than the length of *durations*:

```
>>> notetools.make_notes([0], [(1, 16), (1, 8), (1, 8)])
[Note("c'16"), Note("c'8"), Note("c'8")]
```

Cycle through *durations* when the length of *durations* is less than the length of *pitches*:

```
>>> notetools.make_notes([0, 2, 4, 5, 7], [(1, 16), (1, 8), (1, 8)])
[Note("c'16"), Note("d'8"), Note("e'8"), Note("f'16"), Note("g'8")]
```

Create ad hoc tuplets for nonassignable durations:

```
>>> notetools.make_notes([0], [(1, 16), (1, 12), (1, 8)])
[Note("c'16"), Tuplet(2/3, [c'8]), Note("c'8")]
```

Set `decrease_durations_monotonically=True` to express tied values in decreasing duration:

```
>>> notetools.make_notes([0], [(13, 16)], decrease_durations_monotonically=True)
[Note("c'2."), Note("c'16")]
```

Set `decrease_durations_monotonically=False` to express tied values in increasing duration:

```
>>> notetools.make_notes([0], [(13, 16)], decrease_durations_monotonically=False)
[Note("c'16"), Note("c'2.")]
```

Set *pitches* to a single pitch or a sequence of pitches.

Set *durations* to a single duration or a list of durations.

Return list of newly constructed notes.

19.2.5 notetools.make_notes_with_multiplied_durations

`notetools.`**`make_notes_with_multiplied_durations`**(*pitch*, *written_duration*, *multiplied_durations*)

New in version 2.0. Make *written_duration* notes with *pitch* and *multiplied_durations*:

```
>>> args = [0, Duration(1, 4), [(1, 2), (1, 3), (1, 4), (1, 5)]]
>>> notetools.make_notes_with_multiplied_durations(*args)
[Note("c'4 * 2"), Note("c'4 * 4/3"), Note("c'4 * 1"), Note("c'4 * 4/5")]
```

Useful for making spatially positioned notes.

Return list of notes.

19.2.6 notetools.make_percussion_note

`notetools.`**`make_percussion_note`**(*pitch*, *total_duration*, *max_note_duration=(1, 8)*)

New in version 1.0. Make percussion note:

```
>>> notetools.make_percussion_note(2, (1, 4), (1, 8))
[Note("d'8"), Rest('r8')]
```

```
>>> notetools.make_percussion_note(2, (1, 64), (1, 8))
[Note("d'64")]
```

```
>>> notetools.make_percussion_note(2, (5, 64), (1, 8))
[Note("d'16"), Rest('r64')]
```

```
>>> notetools.make_percussion_note(2, (5, 4), (1, 8))
[Note("d'8"), Rest('r1'), Rest('r8')]
```

Return list of newly constructed note followed by zero or more newly constructed rests.

Durations of note and rests returned will sum to *total_duration*.

Duration of note returned will be no greater than *max_note_duration*.

Duration of rests returned will sum to note duration taken from *total_duration*.

Useful for percussion music where attack duration is negligible and tied notes undesirable.

19.2.7 notetools.make_quarter_notes_with_lilypond_multipliers

notetools.**make_quarter_notes_with_lilypond_multipliers**(*pitches*, *multiplied_durations*)

New in version 2.0. Make quarter notes with *pitches* and *multiplied_durations*:

```
>>> args = [[0, 2, 4, 5], [(1, 4), (1, 5), (1, 6), (1, 7)]]
>>> notetools.make_quarter_notes_with_lilypond_multipliers(*args)
[Note("c'4 * 1"), Note("d'4 * 4/5"), Note("e'4 * 2/3"), Note("f'4 * 4/7")]
```

Read *pitches* cyclically where the length of *pitches* is less than the length of *multiplied_durations*:

```
>>> args = [[0], [(1, 4), (1, 5), (1, 6), (1, 7)]]
>>> notetools.make_quarter_notes_with_lilypond_multipliers(*args)
[Note("c'4 * 1"), Note("c'4 * 4/5"), Note("c'4 * 2/3"), Note("c'4 * 4/7")]
```

Read *multiplied_durations* cyclically where the length of *multiplied_durations* is less than the length of *pitches*:

```
>>> args = [[0, 2, 4, 5], [(1, 5)]]
>>> notetools.make_quarter_notes_with_lilypond_multipliers(*args)
[Note("c'4 * 4/5"), Note("d'4 * 4/5"), Note("e'4 * 4/5"), Note("f'4 * 4/5")]
```

Return list of zero or more newly constructed notes.

19.2.8 notetools.make_repeated_notes

notetools.**make_repeated_notes**(*count*, *duration=Duration(1, 8)*)

Make *count* repeated notes with note head-assignable *duration*:

```
>>> notetools.make_repeated_notes(4)
[Note("c'8"), Note("c'8"), Note("c'8"), Note("c'8")]
```

Make *count* repeated tie chains with tied *duration*:

```
>>> notes = notetools.make_repeated_notes(2, (5, 16))
>>> voice = Voice(notes)
```

```
>>> f(voice)
\new Voice {
    c'4 ~
    c'16
    c'4 ~
    c'16
}
```

Make ad hoc tuplet holding *count* repeated notes with non-power-of-two *duration*:

```
>>> notetools.make_repeated_notes(3, (1, 12))
[Tuplet(2/3, [c'8, c'8, c'8])]
```

Set pitch of all notes created to middle C.

Return list of zero or more newly constructed notes or list of one newly constructed tuplet.

19.2.9 notetools.make_repeated_notes_from_time_signature

notetools.**make_repeated_notes_from_time_signature**(*time_signature*, *pitch="c"'*)

New in version 2.0. Make repeated notes from *time_signature*:

```
>>> notetools.make_repeated_notes_from_time_signature((5, 32))
[Note("c'32"), Note("c'32"), Note("c'32"), Note("c'32"), Note("c'32")]
```

Make repeated notes with *pitch* from *time_signature*:

```
>>> notetools.make_repeated_notes_from_time_signature((5, 32), pitch="d''")
[Note("d''32"), Note("d''32"), Note("d''32"), Note("d''32"), Note("d''32")]
```

Return list of notes.

19.2.10 notetools.make_repeated_notes_from_time_signatures

notetools.**make_repeated_notes_from_time_signatures**(*time_signatures*, *pitch="c"'*)

Make repated notes from *time_signatures*:

```
notetools.make_repeated_notes_from_time_signatures([(2, 8), (3, 32)])
[[Note("c'8"), Note("c'8")], [Note("c'32"), Note("c'32"), Note("c'32")]]
```

Make repeated notes with *pitch* from *time_signatures*:

```
>>> notetools.make_repeated_notes_from_time_signatures([(2, 8), (3, 32)], pitch="d''")
[[Note("d''8"), Note("d''8")], [Note("d''32"), Note("d''32"), Note("d''32")]]
```

Return two-dimensional list of note lists.

Use `sequencetools.flatten_sequence()` to flatten output if required.

19.2.11 notetools.make_repeated_notes_with_shorter_notes_at_end

notetools.**make_repeated_notes_with_shorter_notes_at_end**(*pitch*, *written_duration*, *total_duration*, *prolation=1*)

Make repeated notes with *pitch* and *written_duration* summing to *total_duration* under *prolation*:

```
>>> args = [0, Duration(1, 16), Duration(1, 4)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

```
>>> f(voice)
\new Voice {
    c'16
    c'16
    c'16
    c'16
}
```

Fill power-of-two remaining duration with power-of-two notes of lesser written duration:

```
>>> args = [0, Duration(1, 16), Duration(9, 32)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

```
>>> f(voice)
\new Voice {
    c'16
    c'16
    c'16
    c'16
    c'32
}
```

Fill non-power-of-two remaining duration with ad hoc tuplet:

```
>>> args = [0, Duration(1, 16), Duration(4, 10)]
>>> notes = notetools.make_repeated_notes_with_shorter_notes_at_end(*args)
>>> voice = Voice(notes)
```

```
>>> f(voice)
\new Voice {
    c'16
    c'16
    c'16
    c'16
    c'16
    c'16
    \times 4/5 {
        c'32
    }
}
```

Set *prolation* when making notes in a measure with a non-power-of-two denominator.

Return list of components.

19.2.12 notetools.make_tied_note

`notetools.`**`make_tied_note`**(*pitch*, *duration*, *decrease_durations_monotonically=True*, *forbidden_written_duration=None*)

Returns a list of notes to fill the given duration.

Notes returned are tie-spanned.

19.2.13 notetools.yield_groups_of_notes_in_sequence

`notetools.`**`yield_groups_of_notes_in_sequence`**(*sequence*)

New in version 2.0. Yield groups of notes in *sequence*:

```
>>> staff = Staff("c'8 d'8 r8 r8 <e' g'>8 <f' a'>8 g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    r8
    r8
    <e' g'>8
    <f' a'>8
    g'8
    a'8
    r8
    r8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for note in notetools.yield_groups_of_notes_in_sequence(staff):
...     note
...
(Note("c'8"), Note("d'8"))
(Note("g'8"), Note("a'8"))
```

Return generator.

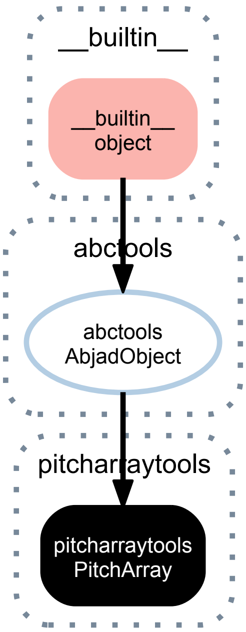
CHAPTER

TWENTY

PITCHARRAYTOOLS

20.1 Concrete Classes

20.1.1 pitcharraytools.PitchArray



class `pitcharraytools.`**`PitchArray`**(**args*)

New in version 2.0. Two-dimensional array of pitches.

Read-only properties

`PitchArray.`**`cell_tokens_by_row`**

`PitchArray.`**`cell_widths_by_row`**

`PitchArray.`**`cells`**

`PitchArray.`**`columns`**

`PitchArray.`**`depth`**

`PitchArray.`**`dimensions`**

`PitchArray.`**`has_voice_crossing`**

`PitchArray.`**`is_rectangular`**

`PitchArray.`**`pitches`**

`PitchArray.`**`pitches_by_row`**

`PitchArray.`**`rows`**

`PitchArray.`**`size`**

`PitchArray.`**`storage_format`**
Storage format of Abjad object.

Return string.

`PitchArray.`**`voice_crossing_count`**

`PitchArray.`**`weight`**

`PitchArray.`**`width`**

Methods

`PitchArray.`**`append_column`**(*column*)

`PitchArray.`**`append_row`**(*row*)

`PitchArray.`**`apply_pitches_by_row`**(*pitch_lists*)

`PitchArray.`**`copy_subarray`**(*upper_left_pair*, *lower_right_pair*)

`PitchArray.`**`has_spanning_cell_over_index`**(*index*)

`PitchArray.`**`pad_to_depth`**(*depth*)

`PitchArray.`**`pad_to_width`**(*width*)

`PitchArray.`**`pop_column`**(*column_index*)

`PitchArray.`**`pop_row`**(*row_index=-1*)

`PitchArray.`**`remove_row`**(*row*)

Special methods

`PitchArray.`**`__add__`**(*arg*)

`PitchArray.`**`__contains__`**(*arg*)

`PitchArray.`**`__copy__`**()

`PitchArray.`**`__eq__`**(*arg*)

`PitchArray.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`PitchArray.`**`__getitem__`**(*arg*)

`PitchArray.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`PitchArray.`**`__iadd__`**(*arg*)

`PitchArray.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`PitchArray.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

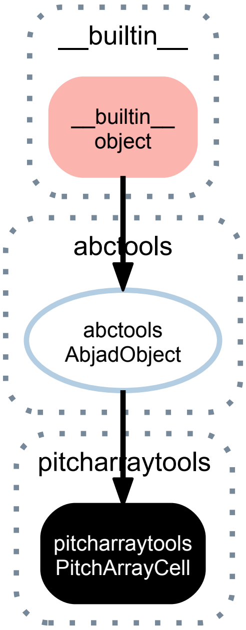
`PitchArray.__ne__(arg)`

`PitchArray.__repr__()`

`PitchArray.__setitem__(i, arg)`

`PitchArray.__str__()`

20.1.2 pitcharraytools.PitchArrayCell



class `pitcharraytools.PitchArrayCell(cell_token=None)`
One cell in a pitch array.

```
>>> array = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> print array
[ ] [     ] [ ]
[     ] [ ] [ ]
>>> cell = array[0][1]
>>> cell
PitchArrayCell(x2)
```

```
>>> cell.column_indices
(1, 2)
```

```
>>> cell.indices
(0, (1, 2))
```

```
>>> cell.is_first_in_row
False
```

```
>>> cell.is_last_in_row
False
```

```
>>> cell.next
PitchArrayCell(x1)
```

```
>>> cell.parent_array
PitchArray(PitchArrayRow(x1, x2, x1), PitchArrayRow(x2, x1, x1))
```

```
>>> cell.parent_column
PitchArrayColumn(x2, x2)
```

```
>>> cell.parent_row
PitchArrayRow(x1, x2, x1)
```

```
>>> cell.pitches
[]
```

```
>>> cell.prev
PitchArrayCell(x1)
```

```
>>> cell.row_index
0
```

```
>>> cell.token
2
```

```
>>> cell.width
2
```

Return pitch array cell.

Read-only properties

`PitchArrayCell.`**`column_indices`**
Read-only tuple of one or more nonnegative integer indices.

`PitchArrayCell.`**`indices`**

`PitchArrayCell.`**`is_first_in_row`**

`PitchArrayCell.`**`is_last_in_row`**

`PitchArrayCell.`**`next`**

`PitchArrayCell.`**`parent_array`**

`PitchArrayCell.`**`parent_column`**

`PitchArrayCell.`**`parent_row`**

`PitchArrayCell.`**`prev`**

`PitchArrayCell.`**`row_index`**

`PitchArrayCell.`**`storage_format`**
Storage format of Abjad object.

Return string.

`PitchArrayCell.`**`token`**

`PitchArrayCell.`**`weight`**

`PitchArrayCell.`**`width`**

Read/write properties

`PitchArrayCell.`**`pitches`**

Methods

`PitchArrayCell.`**`matches_cell`**(*arg*)

Special methods

`PitchArrayCell.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`PitchArrayCell.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PitchArrayCell.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`PitchArrayCell.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PitchArrayCell.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

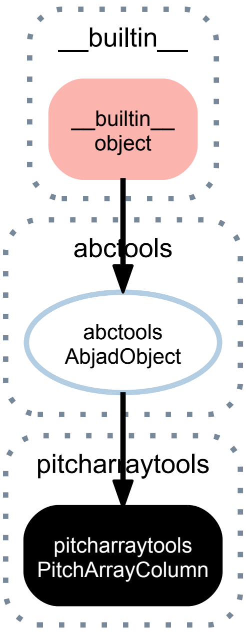
`PitchArrayCell.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`PitchArrayCell.__repr__()`

`PitchArrayCell.__str__()`

20.1.3 pitcharraytools.PitchArrayColumn



class `pitcharraytools.PitchArrayColumn(cells)`
New in version 2.0. Column in a pitch array:

```
>>> array = pitcharraytools.PitchArray([
...    [1, (2, 1), (-1.5, 2)],
...    [(7, 2), (6, 1), 1]])
```

```
>>> print array
[  ] [d'] [bqf    ]
[g'     ] [fs'] [ ]
```

```
>>> array.columns[0]
PitchArrayColumn(x1, g' x2)
```

```
>>> print array.columns[0]
[  ]
[g'     ]
```

Return pitch array column.

Read-only properties

PitchArrayColumn.**cell_tokens**

PitchArrayColumn.**cell_widths**

PitchArrayColumn.**cells**

PitchArrayColumn.**column_index**

PitchArrayColumn.**depth**

PitchArrayColumn.**dimensions**

PitchArrayColumn.**has_voice_crossing**

PitchArrayColumn.**is_defective**

PitchArrayColumn.**parent_array**

PitchArrayColumn.**pitches**

PitchArrayColumn.**start_cells**

PitchArrayColumn.**start_pitches**

PitchArrayColumn.**stop_cells**

PitchArrayColumn.**stop_pitches**

PitchArrayColumn.**storage_format**
Storage format of Abjad object.

Return string.

PitchArrayColumn.**weight**

PitchArrayColumn.**width**

Methods

PitchArrayColumn.**append**(*cell*)

PitchArrayColumn.**extend**(*cells*)

PitchArrayColumn.**remove_pitches**()

Special methods

PitchArrayColumn.**__eq__**(*arg*)

PitchArrayColumn.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PitchArrayColumn.**__getitem__**(*arg*)

PitchArrayColumn.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PitchArrayColumn.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PitchArrayColumn.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

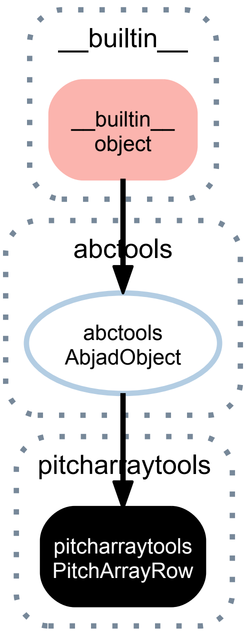
Raise exception.

PitchArrayColumn.**__ne__**(*arg*)

PitchArrayColumn.**__repr__**()

PitchArrayColumn.**__str__**()

20.1.4 pitcharraytools.PitchArrayRow



class pitcharraytools.**PitchArrayRow**(*cells*)
New in version 2.0. One row in pitch array.

```
>>> array = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> array[0].cells[0].pitches.append(0)
>>> array[0].cells[1].pitches.append(2)
>>> array[1].cells[2].pitches.append(4)
>>> print array
[c'] [d'     ] [  ]
[          ] [ ] [e']
```

```
>>> array[0]
PitchArrayRow(c', d' x2, x1)
```

```
>>> array[0].cell_widths
(1, 2, 1)
```

```
>>> array[0].dimensions
(1, 4)
```

```
>>> array[0].pitches
(NamedChromaticPitch("c'"), NamedChromaticPitch("d'"))
```

Return pitch array row.

Read-only properties

PitchArrayRow.**cell_tokens**

PitchArrayRow.**cell_widths**

PitchArrayRow.**cells**

PitchArrayRow.**depth**

PitchArrayRow.**dimensions**

PitchArrayRow.**is_defective**

PitchArrayRow.**is_in_range**

PitchArrayRow.**parent_array**

PitchArrayRow.**pitches**

PitchArrayRow.**row_index**

PitchArrayRow.**storage_format**
Storage format of Abjad object.

Return string.

PitchArrayRow.**weight**

PitchArrayRow.**width**

Read/write properties

PitchArrayRow.**pitch_range**

Methods

PitchArrayRow.**append**(*cell_token*)

PitchArrayRow.**apply_pitches**(*pitch_tokens*)

PitchArrayRow.**copy_subrow**(*start=None*, *stop=None*)

PitchArrayRow.**empty_pitches**()

PitchArrayRow.**extend**(*cell_tokens*)

PitchArrayRow.**has_spanning_cell_over_index**(*i*)

PitchArrayRow.**index**(*cell*)

PitchArrayRow.**merge**(*cells*)

PitchArrayRow.**pad_to_width**(*width*)

PitchArrayRow.**pop**(*cell_index*)

PitchArrayRow.**remove**(*cell*)

PitchArrayRow.**withdraw**()

Special methods

PitchArrayRow.**__add__**(*arg*)

PitchArrayRow.**__copy__**()

PitchArrayRow.**__eq__**(*arg*)

PitchArrayRow.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PitchArrayRow.**__getitem__**(*arg*)

PitchArrayRow.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PitchArrayRow.**__iadd__**(*arg*)

PitchArrayRow.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PitchArrayRow.**__len__**()

PitchArrayRow.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PitchArrayRow.**__ne__**(*arg*)

PitchArrayRow.**__repr__**()

PitchArrayRow.**__str__**()

20.2 Functions

20.2.1 pitcharraytools.all_are_pitch_arrays

pitcharraytools.**all_are_pitch_arrays**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad pitch arrays:

```
>>> pitch_array = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
```

```
>>> pitcharraytools.all_are_pitch_arrays([pitch_array])
True
```

True when *expr* is an empty sequence:

```
>>> pitcharraytools.all_are_pitch_arrays([])
True
```

Otherwise false:

```
>>> pitcharraytools.all_are_pitch_arrays('foo')
False
```

Return boolean.

20.2.2 pitcharraytools.concatenate_pitch_arrays

pitcharraytools.**concatenate_pitch_arrays**(*pitch_arrays*)
New in version 2.0. Concatenate *pitch_arrays*:

```
>>> array_1 = pitcharraytools.PitchArray([[1, 2, 1], [2, 1, 1]])
>>> print array_1
[ ] [     ] [ ]
[     ] [ ] [ ]
```

```
>>> array_2 = pitcharraytools.PitchArray([[3, 4], [4, 3]])
>>> print array_2
[     ] [           ]
[           ] [     ]
```

```
>>> array_3 = pitcharraytools.PitchArray([[1, 1], [1, 1]])
>>> print array_3
[ ] [ ]
[ ] [ ]
```

```
>>> merged_array = pitcharraytools.concatenate_pitch_arrays([array_1, array_2, array_3])
>>> print merged_array
[ ] [     ] [ ] [     ] [           ] [ ] [ ]
[     ] [ ] [ ] [           ] [     ] [ ] [ ]
```

Return pitch array.

20.2.3 pitcharraytools.list_nonspanning_subarrays_of_pitch_array

pitcharraytools.**list_nonspanning_subarrays_of_pitch_array**(*pitch_array*)
New in version 2.0. List nonspanning subarrays of *pitch_array*:

```
>>> array = pitcharraytools.PitchArray([
...     [2, 2, 3, 1],
...     [1, 2, 1, 1, 2, 1],
...     [1, 1, 1, 1, 1, 1, 1, 1]])
>>> print array
[     ] [     ] [           ] [ ]
[ ] [     ] [ ] [ ] [     ] [ ]
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

```
>>> subarrays = pitcharraytools.list_nonspanning_subarrays_of_pitch_array(array)
>>> len(subarrays)
3
```

```
>>> print subarrays[0]
[     ] [     ]
[ ] [     ] [ ]
[ ] [ ] [ ] [ ]
```

```
>>> print subarrays[1]
[           ]
[ ] [     ]
[ ] [ ] [ ]
```

```
>>> print subarrays[2]
[ ]
[ ]
[ ]
```

Return list.

20.2.4 pitcharraytools.make_empty_pitch_array_from_list_of_pitch_lists

pitcharraytools.**make_empty_pitch_array_from_list_of_pitch_lists**(*leaf_iterables*)
New in version 2.0. Make empty pitch array from *leaf_iterables*:

```
>>> score = Score([])
>>> score.append(Staff("c'8 d'8 e'8 f'8"))
>>> score.append(Staff("c'4 d'4"))
>>> score.append(Staff(tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8") * 2))
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        d'4
    }
    \new Staff {
        \times 2/3 {
            c'8
            d'8
            e'8
        }
        \times 2/3 {
            c'8
            d'8
            e'8
        }
    }
>>
```

```
>>> array = pitcharraytools.make_empty_pitch_array_from_list_of_pitch_lists(score)
>>> print array
[     ] [     ] [     ] [     ]
[                 ] [                 ]
[ ] [     ] [ ] [ ] [     ] [ ]
```

Return pitch array.

20.2.5 pitcharraytools.make_pitch_array_score_from_pitch_arrays

pitcharraytools.**make_pitch_array_score_from_pitch_arrays**(*pitch_arrays*)

New in version 2.0. Make pitch-array score from *pitch_arrays*:

```
>>> array_1 = pitcharraytools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> array_2 = pitcharraytools.PitchArray([
...     [1, 1, 1],
...     [1, 1, 1]])
```

```
>>> score = pitcharraytools.make_pitch_array_score_from_pitch_arrays([array_1, array_2])
```

```
>>> f(score)
\new Score <<
    \new StaffGroup <<
        \new Staff {
            {
                \time 4/8
                r8
                d'8
                <bf bqf>4
            }
            {
                \time 3/8
                r8
                r8
                r8
```

```
                }
            }
            \new Staff {
                {
                    \time 4/8
                    g'4
                    fs'8
                    r8
                }
                {
                    \time 3/8
                    r8
                    r8
                    r8
                }
            }
        >>
    >>
```

Create one staff per pitch-array row.

Return score.

20.2.6 pitcharraytools.make_populated_pitch_array_from_list_of_pitch_lists

pitcharraytools.**make_populated_pitch_array_from_list_of_pitch_lists**(*leaf_iterables*)
New in version 2.0. Make populated pitch array from *leaf_iterables*:

```
>>> score = Score([])
>>> score.append(Staff("c'8 d'8 e'8 f'8"))
>>> score.append(Staff("c'4 d'4"))
>>> score.append(Staff(tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8") * 2))
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
    \new Staff {
        c'4
        d'4
    }
    \new Staff {
        \times 2/3 {
            c'8
            d'8
            e'8
        }
        \times 2/3 {
            c'8
            d'8
            e'8
        }
    }
>>
```

```
>>> array = pitcharraytools.make_populated_pitch_array_from_list_of_pitch_lists(score)
>>> print array
[c'     ] [d'     ] [e'     ] [f'     ]
[c'                   ] [d'                   ]
[c'] [d'     ] [e'] [c'] [d'     ] [e']
```

Return pitch array.

20.2.7 pitcharraytools.pitch_array_row_to_measure

pitcharraytools.**pitch_array_row_to_measure**(*pitch_array_row*, *cell_duration_denominator=8*)

New in version 2.0. Change *pitch_array_row* to measure with time signature *pitch_array_row.width* over *cell_duration_denominator*:

```
>>> array = pitcharraytools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> print array
[  ] [d'] [bf bqf    ]
[g'     ] [fs'   ] [ ]
```

```
>>> measure = pitcharraytools.pitch_array_row_to_measure(array.rows[0])
```

```
>>> f(measure)
{
    \time 4/8
    r8
    d'8
    <bf bqf>4
}
```

Return measure.

20.2.8 pitcharraytools.pitch_array_to_measures

pitcharraytools.**pitch_array_to_measures**(*pitch_array*, *cell_duration_denominator=8*)

New in version 2.0. Change *pitch_array* to measures with time signatures *row.width* over *cell_duration_denominator* for each row in *pitch_array*:

```
>>> array = pitcharraytools.PitchArray([
...     [1, (2, 1), ([-2, -1.5], 2)],
...     [(7, 2), (6, 1), 1]])
```

```
>>> print array
[  ] [d'] [bf bqf    ]
[g'     ] [fs'   ] [ ]
```

```
>>> pitcharraytools.pitch_array_to_measures(array)
[Measure(4/8, [r8, d'8, <bf bqf>4]), Measure(4/8, [g'4, fs'8, r8])]
>>> for measure in _:
...     f(measure)
...
{
    \time 4/8
    r8
    d'8
    <bf bqf>4
}
{
    \time 4/8
    g'4
    fs'8
    r8
}
```

Return list of measures.

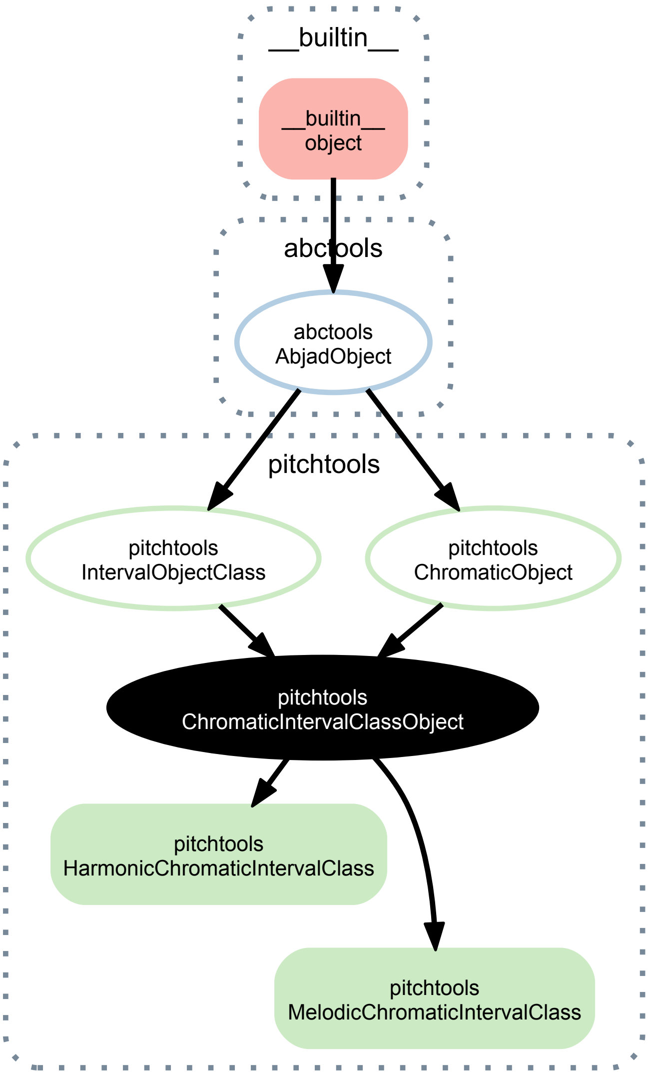
CHAPTER

TWENTYONE

PITCHTOOLS

21.1 Abstract Classes

21.1.1 pitchtools.ChromaticIntervalClassObject



class `pitchtools.`**`ChromaticIntervalClassObject`**

New in version 2.0. Chromatic interval-class base class.

Read-only properties

`ChromaticIntervalClassObject.`**number**

`ChromaticIntervalClassObject.`**storage_format**
Storage format of Abjad object.

Return string.

Special methods

`ChromaticIntervalClassObject.`**__abs__**()

`ChromaticIntervalClassObject.`**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`ChromaticIntervalClassObject.`**__float__**()

`ChromaticIntervalClassObject.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ChromaticIntervalClassObject.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`ChromaticIntervalClassObject.`**__hash__**()

`ChromaticIntervalClassObject.`**__int__**()

`ChromaticIntervalClassObject.`**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ChromaticIntervalClassObject.`**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

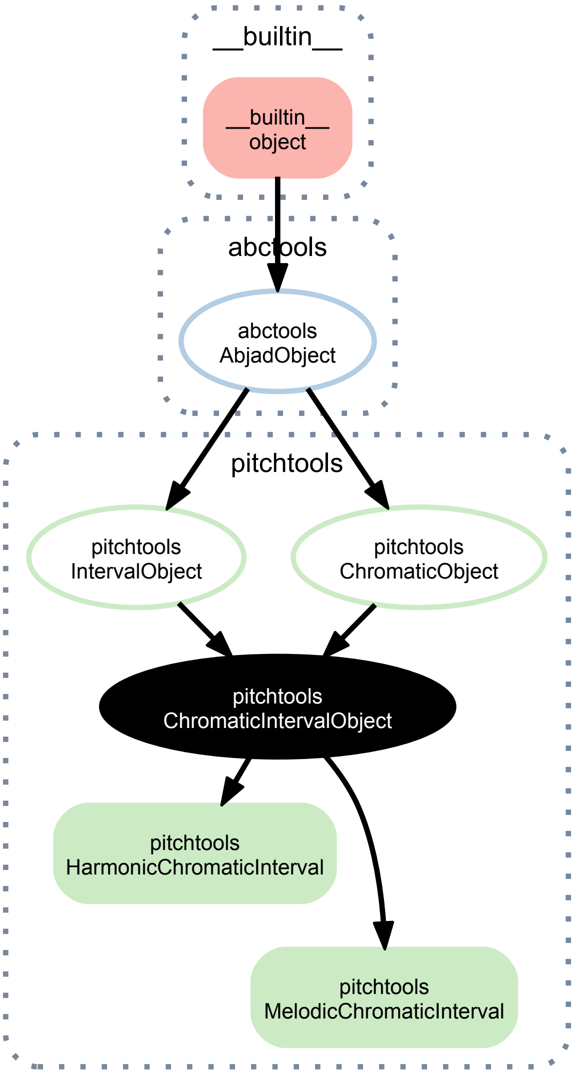
`ChromaticIntervalClassObject.`**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`ChromaticIntervalClassObject.`**__repr__**()

`ChromaticIntervalClassObject.`**__str__**()

21.1.2 pitchtools.ChromaticIntervalObject



class `pitchtools.`**`ChromaticIntervalObject`**(*arg*)

New in version 2.0. Chromatic interval base class.

Read-only properties

`ChromaticIntervalObject.`**`cents`**

`ChromaticIntervalObject.`**`interval_class`**

`ChromaticIntervalObject.`**`number`**

`ChromaticIntervalObject.`**`semitones`**

`ChromaticIntervalObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`ChromaticIntervalObject.`**`__abs__`**()

`ChromaticIntervalObject.`**`__add__`**(*arg*)

ChromaticIntervalObject.**__copy__**()

ChromaticIntervalObject.**__eq__**(*arg*)

ChromaticIntervalObject.**__float__**()

ChromaticIntervalObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ChromaticIntervalObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ChromaticIntervalObject.**__hash__**()

ChromaticIntervalObject.**__int__**()

ChromaticIntervalObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ChromaticIntervalObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

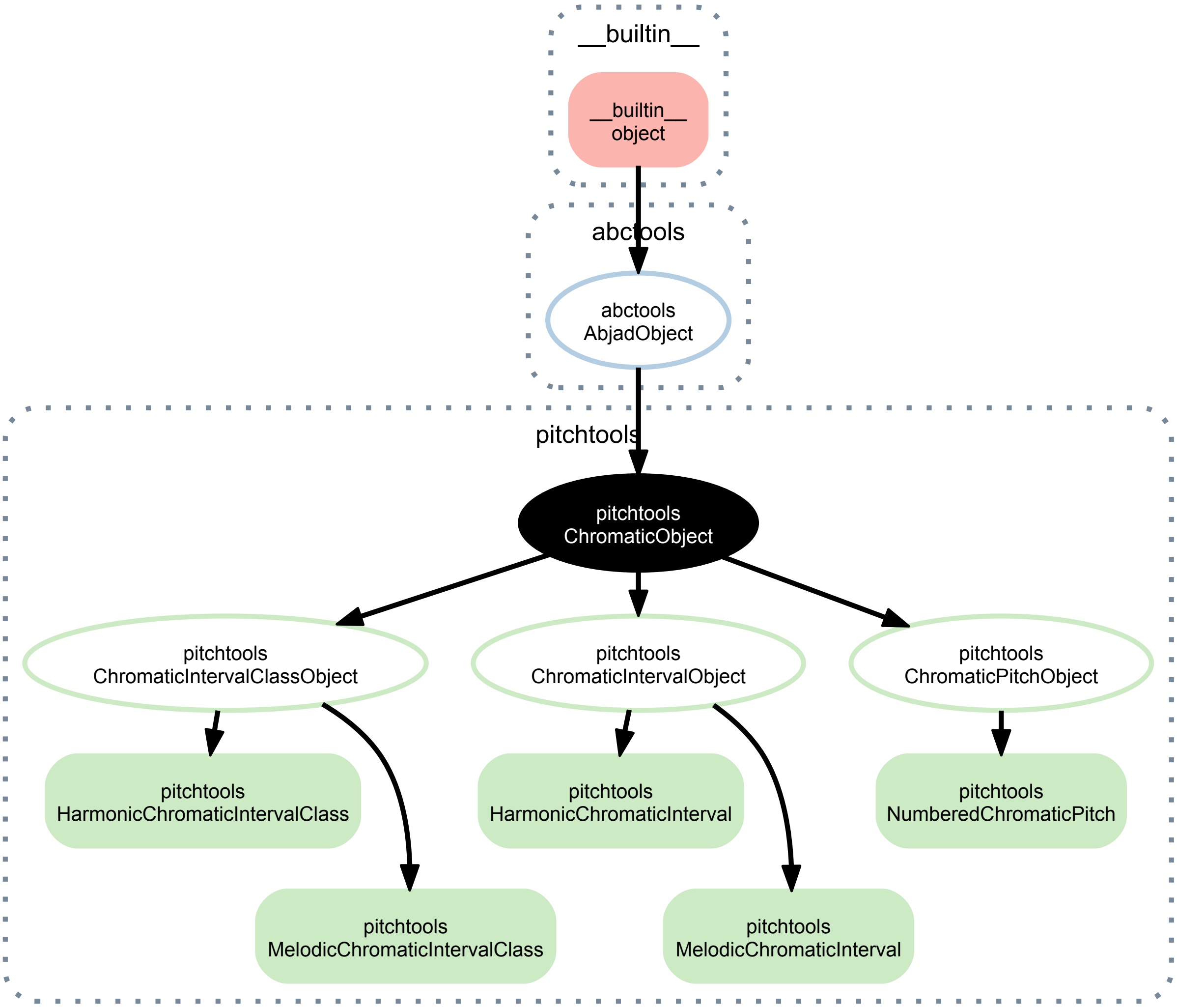
ChromaticIntervalObject.**__ne__**(*arg*)

ChromaticIntervalObject.**__repr__**()

ChromaticIntervalObject.**__str__**()

ChromaticIntervalObject.**__sub__**(*arg*)

21.1.3 pitchtools.ChromaticObject



class `pitchtools.`**`ChromaticObject`**

..versionadded:: 2.0

Chromatic object base class.

Read-only properties

`ChromaticObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`ChromaticObject.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`ChromaticObject.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`ChromaticObject.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`ChromaticObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ChromaticObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

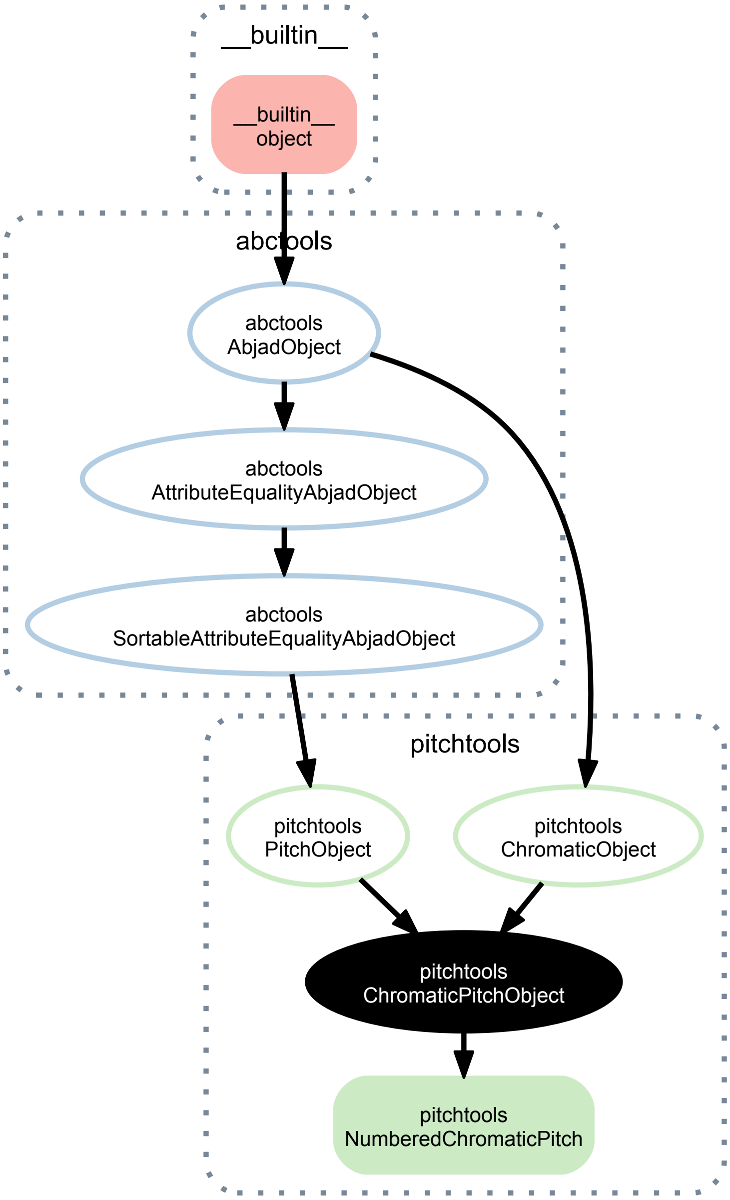
`ChromaticObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`ChromaticObject.__repr__()`
Interpreter representation of Abjad object.

Return string.

21.1.4 pitchtools.ChromaticPitchObject



class `pitchtools.`**`ChromaticPitchObject`**

New in version 2.0. Chromatic pitch base class.

Read-only properties

`ChromaticPitchObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`ChromaticPitchObject.`**`__abs__`**`()`

`ChromaticPitchObject.`**`__eq__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`ChromaticPitchObject.`**`__float__`**`()`

`ChromaticPitchObject.`**`__ge__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`ChromaticPitchObject.`**`__gt__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`ChromaticPitchObject.`**`__hash__`**`()`

`ChromaticPitchObject.`**`__int__`**`()`

`ChromaticPitchObject.`**`__le__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

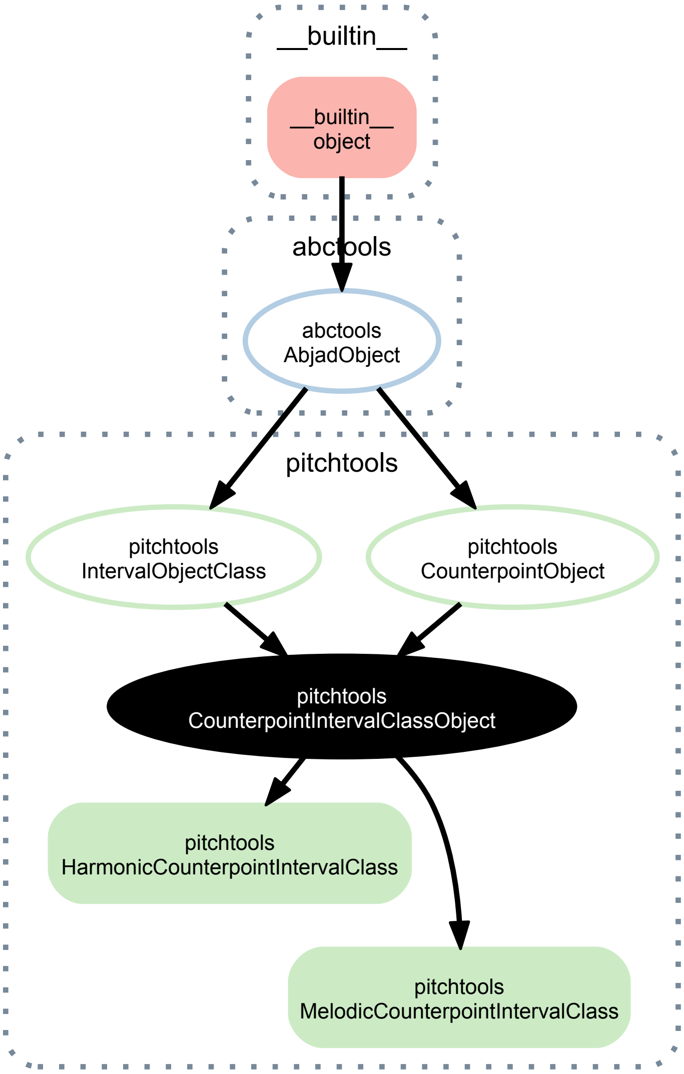
`ChromaticPitchObject.`**`__lt__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`ChromaticPitchObject.`**`__ne__`**`(`*arg*`)`

`ChromaticPitchObject.`**`__repr__`**`()`

21.1.5 pitchtools.CounterpointIntervalClassObject



class `pitchtools.`**`CounterpointIntervalClassObject`**

New in version 2.0. Counterpoint interval-class base class.

Read-only properties

`CounterpointIntervalClassObject.`**`number`**

`CounterpointIntervalClassObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`CounterpointIntervalClassObject.`**`__abs__`**`()`

`CounterpointIntervalClassObject.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`CounterpointIntervalClassObject.`**`__float__`**`()`

`CounterpointIntervalClassObject.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`CounterpointIntervalClassObject.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`CounterpointIntervalClassObject.__hash__()`

`CounterpointIntervalClassObject.__int__()`

`CounterpointIntervalClassObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`CounterpointIntervalClassObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

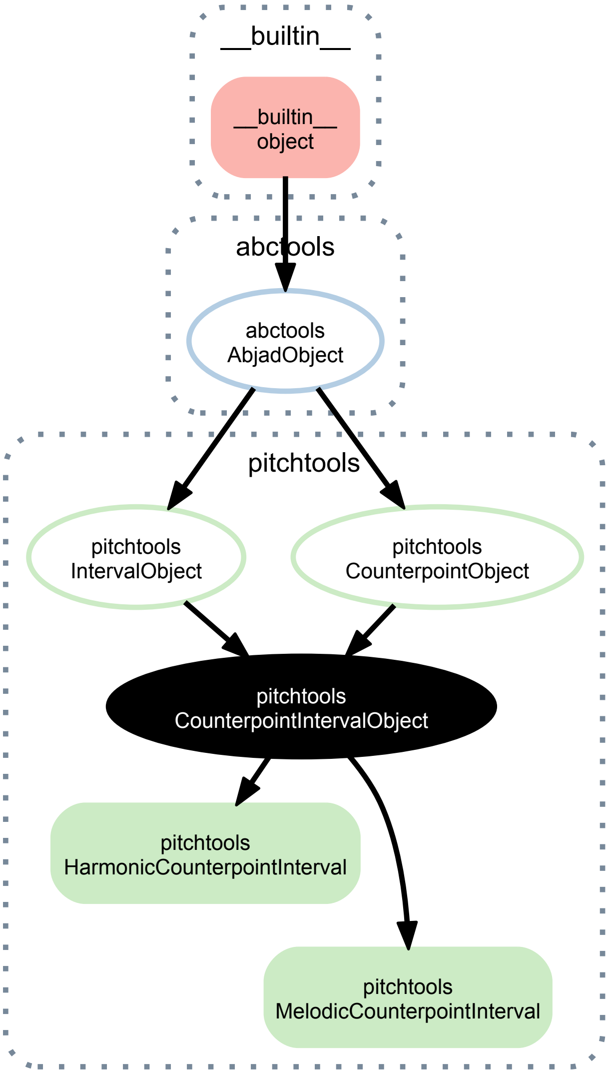
`CounterpointIntervalClassObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`CounterpointIntervalClassObject.__repr__()`

`CounterpointIntervalClassObject.__str__()`

21.1.6 pitchtools.CounterpointIntervalObject



class `pitchtools.`**`CounterpointIntervalObject`**

..versionadded:: 2.0

Counterpoint interval base class.

Read-only properties

`CounterpointIntervalObject.`**`cents`**

`CounterpointIntervalObject.`**`interval_class`**

`CounterpointIntervalObject.`**`number`**

`CounterpointIntervalObject.`**`semitones`**

`CounterpointIntervalObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`CounterpointIntervalObject.`**`__abs__`**`()`

`CounterpointIntervalObject.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`CounterpointIntervalObject.`**`__float__`**`()`

`CounterpointIntervalObject.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`CounterpointIntervalObject.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`CounterpointIntervalObject.`**`__hash__`**`()`

`CounterpointIntervalObject.`**`__int__`**`()`

`CounterpointIntervalObject.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`CounterpointIntervalObject.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

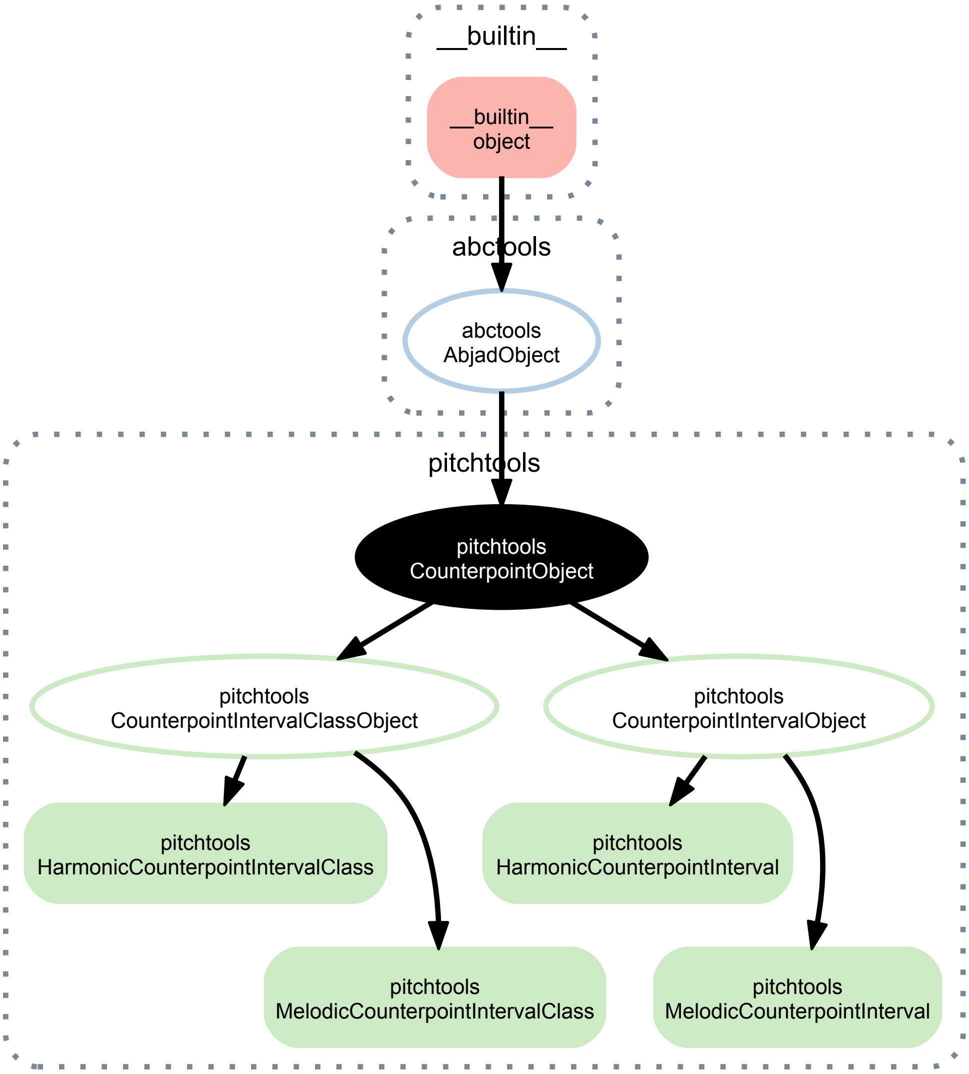
`CounterpointIntervalObject.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`CounterpointIntervalObject.`**`__repr__`**`()`

`CounterpointIntervalObject.`**`__str__`**`()`

21.1.7 pitchtools.CounterpointObject



class `pitchtools.`**`CounterpointObject`**

Counterpoint object base class.

Read-only properties

`CounterpointObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`CounterpointObject.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`CounterpointObject.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

CounterpointObject.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

CounterpointObject.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

CounterpointObject.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

CounterpointObject.**__ne__**(*expr*)

Defined equal to the opposite of equality.

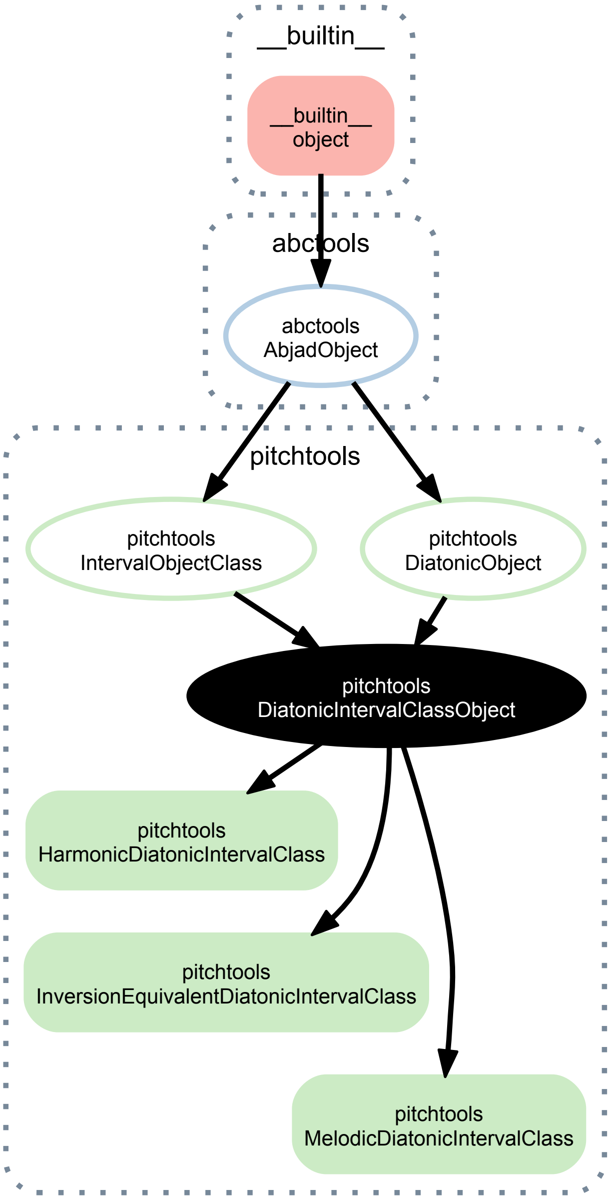
Return boolean.

CounterpointObject.**__repr__**()

Interpreter representation of Abjad object.

Return string.

21.1.8 pitchtools.DiatonicIntervalClassObject



class `pitchtools.`**`DiatonicIntervalClassObject`**

New in version 2.0. Diatonic interval-class base class.

Read-only properties

`DiatonicIntervalClassObject.`**`number`**

`DiatonicIntervalClassObject.`**`quality_string`**

`DiatonicIntervalClassObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiatonicIntervalClassObject.`**`__abs__`**`()`

`DiatonicIntervalClassObject.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`DiatonicIntervalClassObject.`**`__float__`**`()`

`DiatonicIntervalClassObject.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DiatonicIntervalClassObject.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`DiatonicIntervalClassObject.`**`__hash__`**`()`

`DiatonicIntervalClassObject.`**`__int__`**`()`

`DiatonicIntervalClassObject.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DiatonicIntervalClassObject.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

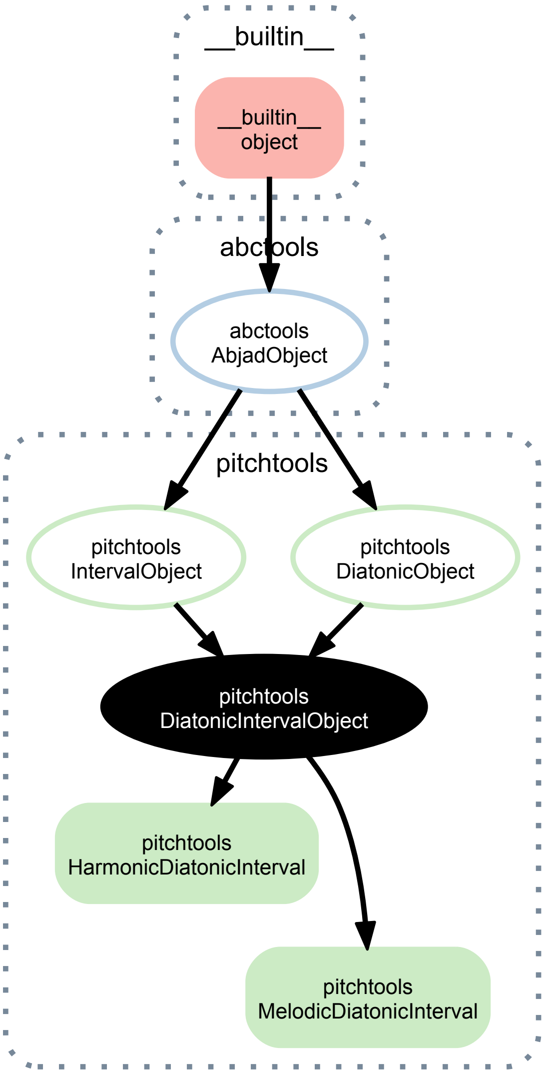
`DiatonicIntervalClassObject.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`DiatonicIntervalClassObject.`**`__repr__`**`()`

`DiatonicIntervalClassObject.`**`__str__`**`()`

21.1.9 pitchtools.DiatonicIntervalObject



class `pitchtools.`**`DiatonicIntervalObject`**(*quality_string*, *number*)

New in version 2.0. Diatonic interval base class.

Read-only properties

`DiatonicIntervalObject.`**`cents`**

`DiatonicIntervalObject.`**`diatonic_interval_class`**

`DiatonicIntervalObject.`**`interval_class`**

`DiatonicIntervalObject.`**`interval_string`**

`DiatonicIntervalObject.`**`number`**

`DiatonicIntervalObject.`**`quality_string`**

`DiatonicIntervalObject.`**`semitones`**

`DiatonicIntervalObject.`**`staff_spaces`**

`DiatonicIntervalObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiatonicIntervalObject.`**`__abs__`**`()`

`DiatonicIntervalObject.`**`__eq__`**`(`*arg*`)`

`DiatonicIntervalObject.`**`__float__`**`()`

`DiatonicIntervalObject.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicIntervalObject.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`DiatonicIntervalObject.`**`__hash__`**`()`

`DiatonicIntervalObject.`**`__int__`**`()`

`DiatonicIntervalObject.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicIntervalObject.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

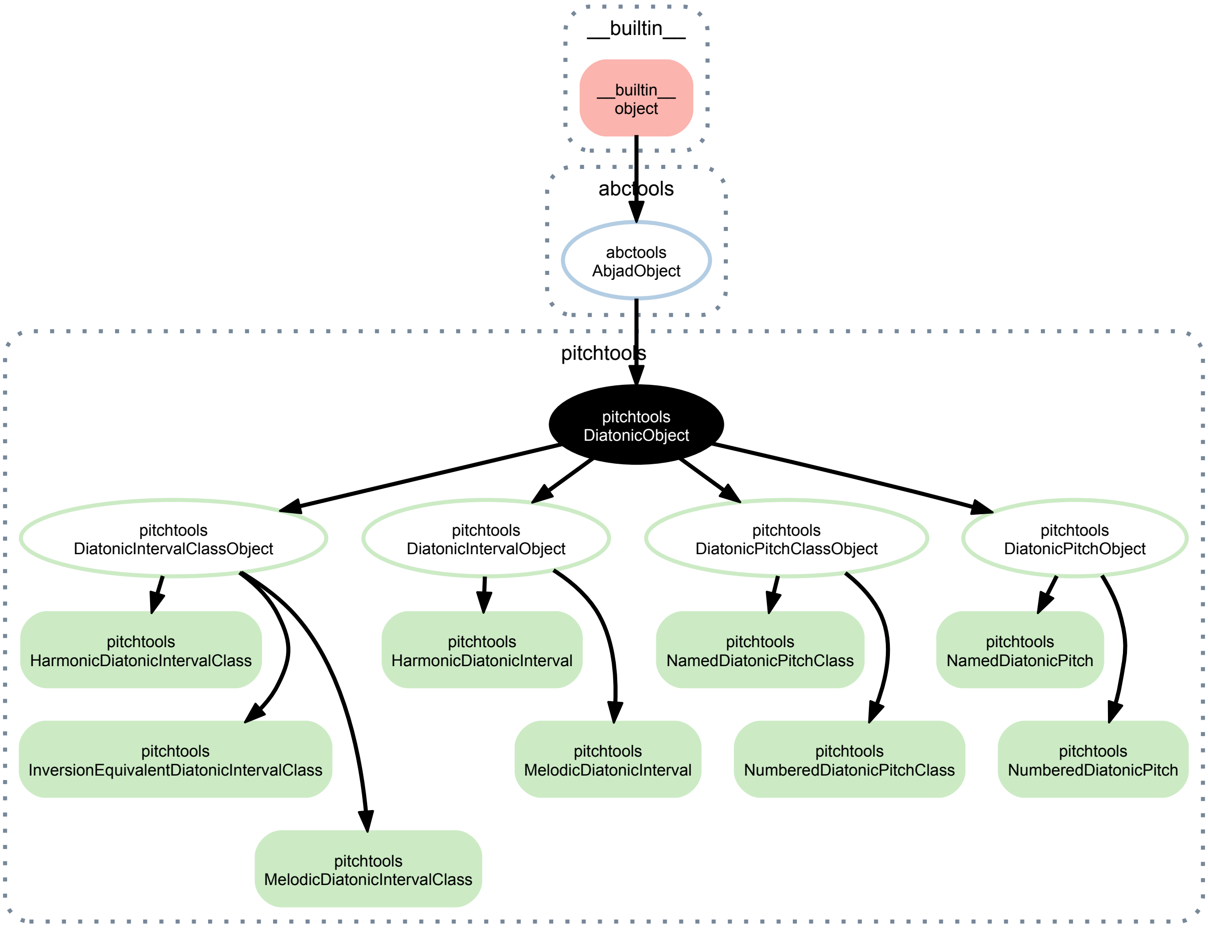
Raise exception.

`DiatonicIntervalObject.`**`__ne__`**`(`*arg*`)`

`DiatonicIntervalObject.`**`__repr__`**`()`

`DiatonicIntervalObject.`**`__str__`**`()`

21.1.10 pitchtools.DiatonicObject



class `pitchtools.`**`DiatonicObject`**

..versionadded:: 2.0

Diatonic object base class.

Read-only properties

`DiatonicObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiatonicObject.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`DiatonicObject.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicObject.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`DiatonicObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DiatonicObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

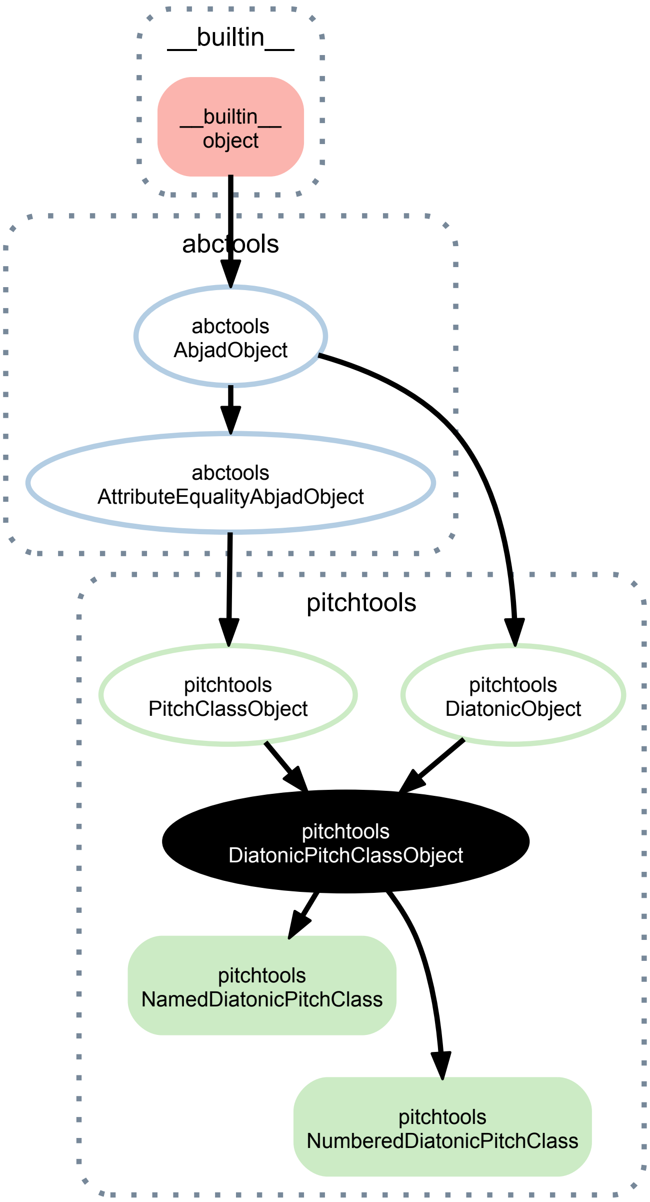
`DiatonicObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`DiatonicObject.__repr__()`
Interpreter representation of Abjad object.

Return string.

21.1.11 pitchtools.DiatonicPitchClassObject



class `pitchtools.`**`DiatonicPitchClassObject`**

New in version 2.0. Diatonic pitch-class base class.

Read-only properties

`DiatonicPitchClassObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiatonicPitchClassObject.`**`__abs__`**`()`

`DiatonicPitchClassObject.`**`__eq__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`DiatonicPitchClassObject.`**`__float__`**`()`

`DiatonicPitchClassObject.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicPitchClassObject.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`DiatonicPitchClassObject.`**`__hash__`**`()`

`DiatonicPitchClassObject.`**`__int__`**`()`

`DiatonicPitchClassObject.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicPitchClassObject.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiatonicPitchClassObject.`**`__ne__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

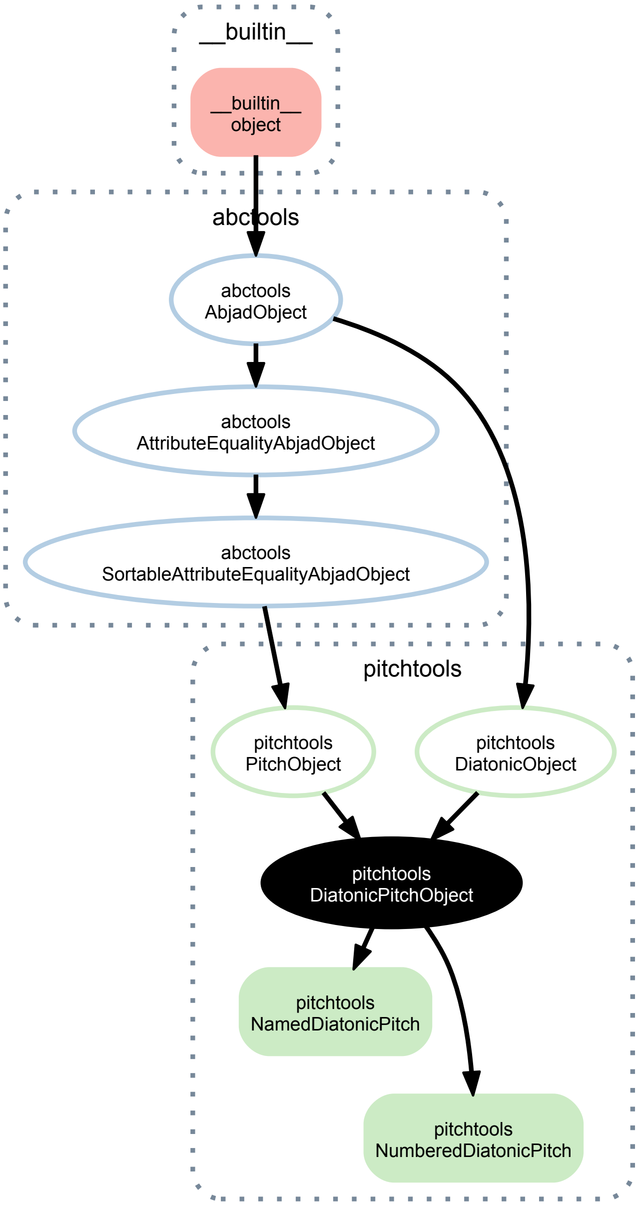
Return boolean.

`DiatonicPitchClassObject.`**`__repr__`**`()`

Interpreter representation of object defined equal to class name and format string.

Return string.

21.1.12 pitchtools.DiatonicPitchObject



class `pitchtools.`**`DiatonicPitchObject`**

New in version 2.0. Diatonic pitch base class.

Read-only properties

`DiatonicPitchObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiatonicPitchObject.`**`__abs__`**`()`

`DiatonicPitchObject.`**`__eq__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

DiatonicPitchObject.**__float__**()

DiatonicPitchObject.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

DiatonicPitchObject.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

DiatonicPitchObject.**__hash__**()

DiatonicPitchObject.**__int__**()

DiatonicPitchObject.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

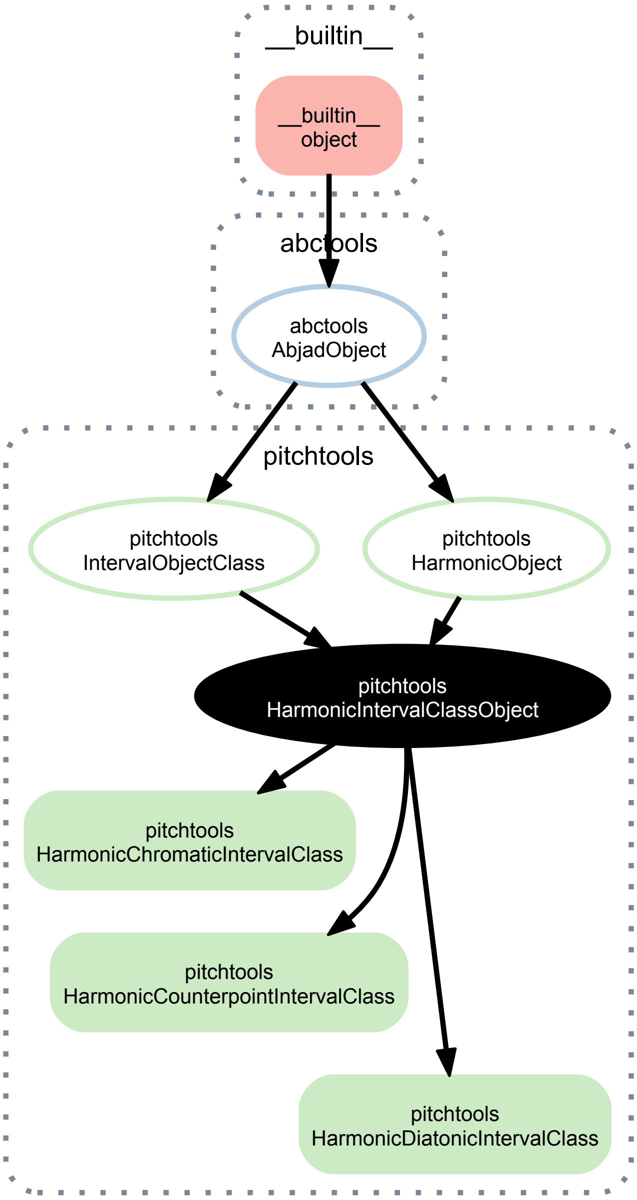
DiatonicPitchObject.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

DiatonicPitchObject.**__ne__**(*arg*)

DiatonicPitchObject.**__repr__**()

21.1.13 pitchtools.HarmonicIntervalClassObject



class `pitchtools.`**`HarmonicIntervalClassObject`**

New in version 2.0. Harmonic interval-class base class.

Read-only properties

`HarmonicIntervalClassObject.`**`number`**

`HarmonicIntervalClassObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`HarmonicIntervalClassObject.`**`__abs__`**`()`

`HarmonicIntervalClassObject.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`HarmonicIntervalClassObject.__float__()`

`HarmonicIntervalClassObject.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`HarmonicIntervalClassObject.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`HarmonicIntervalClassObject.__hash__()`

`HarmonicIntervalClassObject.__int__()`

`HarmonicIntervalClassObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`HarmonicIntervalClassObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

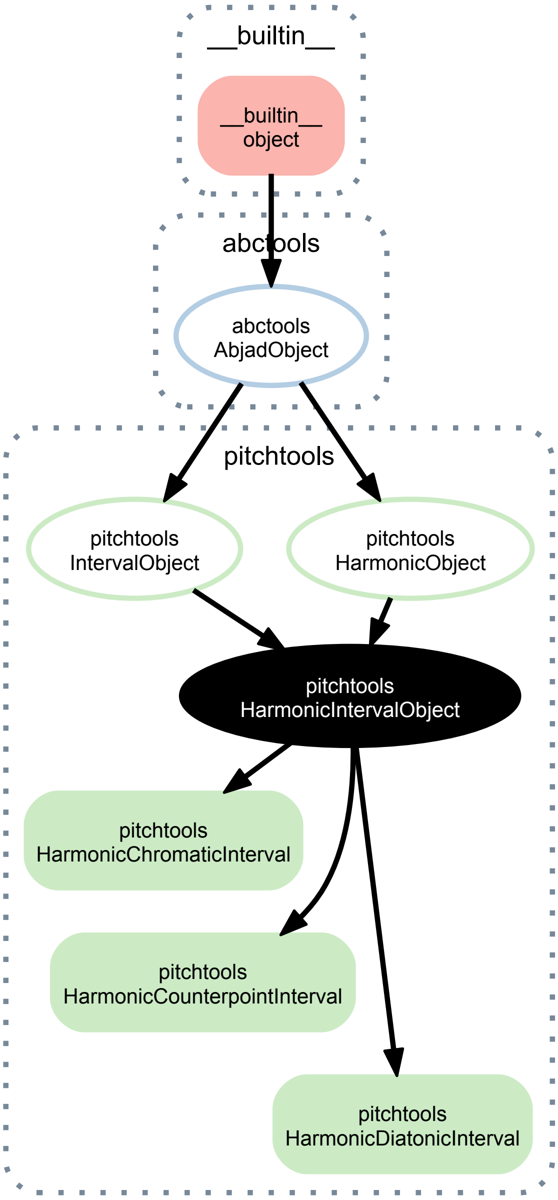
`HarmonicIntervalClassObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`HarmonicIntervalClassObject.__repr__()`

`HarmonicIntervalClassObject.__str__()`

21.1.14 pitchtools.HarmonicIntervalObject



class `pitchtools.`**`HarmonicIntervalObject`**

..versionadded:: 2.0

Harmonic interval base class.

Read-only properties

`HarmonicIntervalObject.`**`cents`**

`HarmonicIntervalObject.`**`interval_class`**

`HarmonicIntervalObject.`**`number`**

`HarmonicIntervalObject.`**`semitones`**

`HarmonicIntervalObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

HarmonicIntervalObject.**__abs__**()

HarmonicIntervalObject.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

HarmonicIntervalObject.**__float__**()

HarmonicIntervalObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicIntervalObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HarmonicIntervalObject.**__hash__**()

HarmonicIntervalObject.**__int__**()

HarmonicIntervalObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicIntervalObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

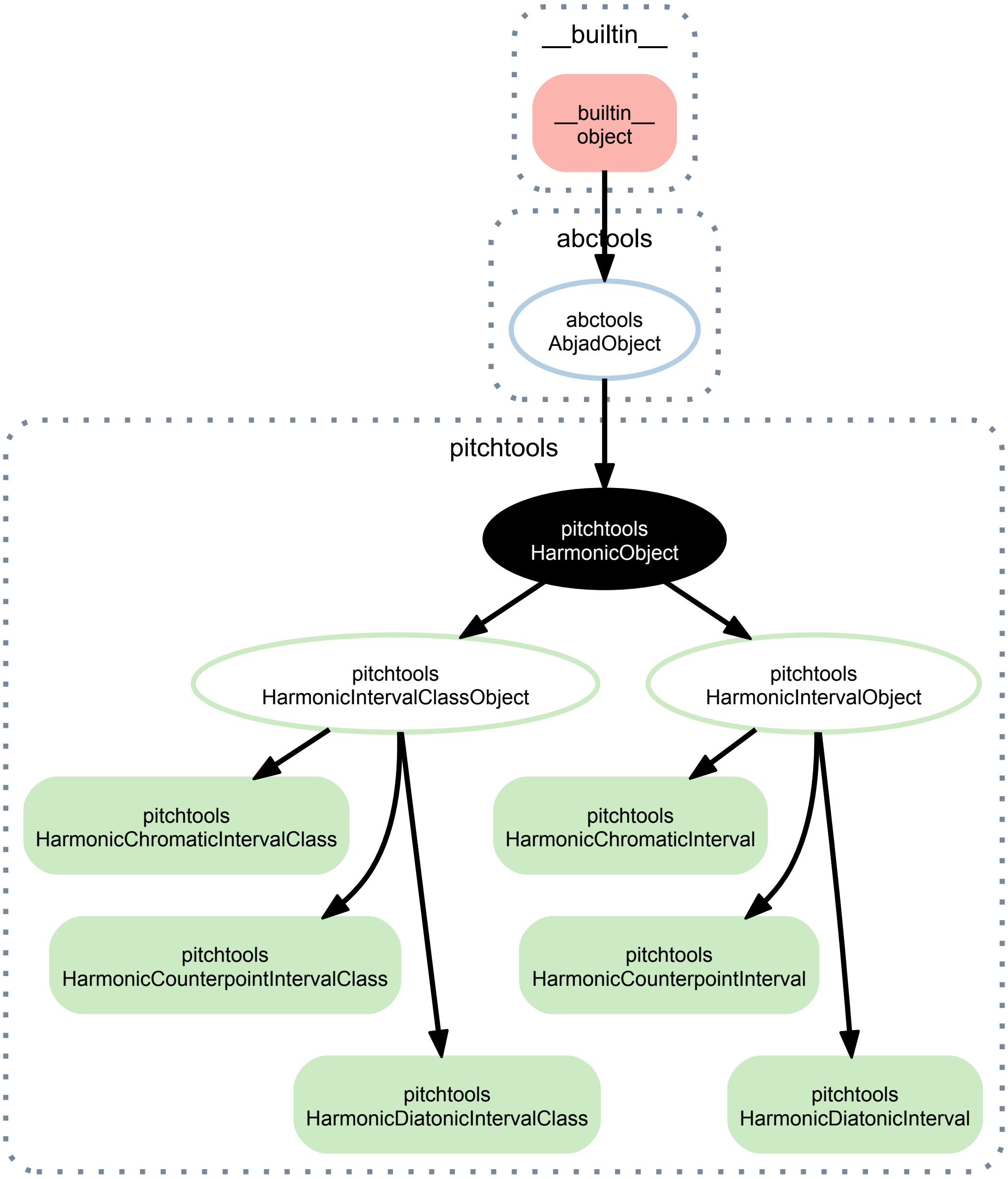
HarmonicIntervalObject.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

HarmonicIntervalObject.**__repr__**()

HarmonicIntervalObject.**__str__**()

21.1.15 pitchtools.HarmonicObject



class `pitchtools.`**`HarmonicObject`**

..versionadded:: 2.0

Harmonic object base class.

Read-only properties

`HarmonicObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`HarmonicObject.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

HarmonicObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HarmonicObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

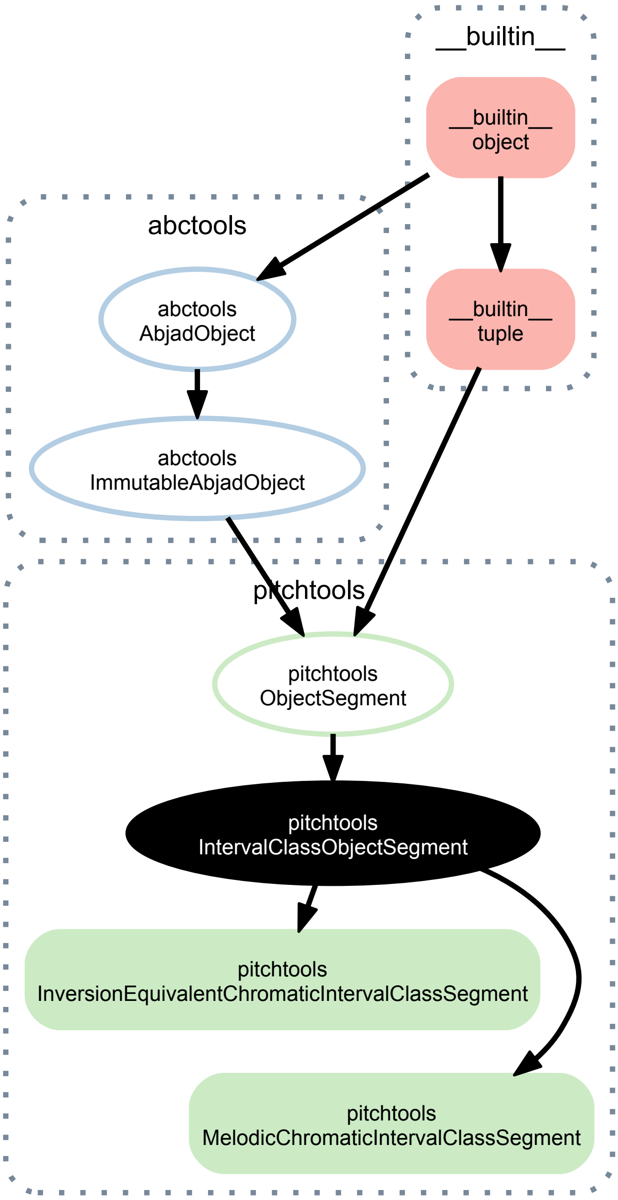
HarmonicObject.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

HarmonicObject.**__repr__**()
Interpreter representation of Abjad object.

Return string.

21.1.16 pitchtools.IntervalClassObjectSegment



class `pitchtools.`**`IntervalClassObjectSegment`**(**args*, ***kwargs*)

New in version 2.0. Interval-class segment base class.

Read-only properties

`IntervalClassObjectSegment.`**`interval_class_numbers`**

`IntervalClassObjectSegment.`**`interval_classes`**

`IntervalClassObjectSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`IntervalClassObjectSegment.`**`count`**(*value*) → integer – return number of occurrences of value

IntervalClassObjectSegment.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Special methods

IntervalClassObjectSegment.**__add__**(*arg*)

IntervalClassObjectSegment.**__contains__**()

x.__contains__(y) <==> y in x

IntervalClassObjectSegment.**__eq__**()

x.__eq__(y) <==> x==y

IntervalClassObjectSegment.**__ge__**()

x.__ge__(y) <==> x>=y

IntervalClassObjectSegment.**__getitem__**()

x.__getitem__(y) <==> x[y]

IntervalClassObjectSegment.**__getslice__**(*start*, *stop*)

IntervalClassObjectSegment.**__gt__**()

x.__gt__(y) <==> x>y

IntervalClassObjectSegment.**__hash__**() <==> hash(x)

IntervalClassObjectSegment.**__iter__**() <==> iter(x)

IntervalClassObjectSegment.**__le__**()

x.__le__(y) <==> x<=y

IntervalClassObjectSegment.**__len__**() <==> len(x)

IntervalClassObjectSegment.**__lt__**()

x.__lt__(y) <==> x<y

IntervalClassObjectSegment.**__mul__**(*n*)

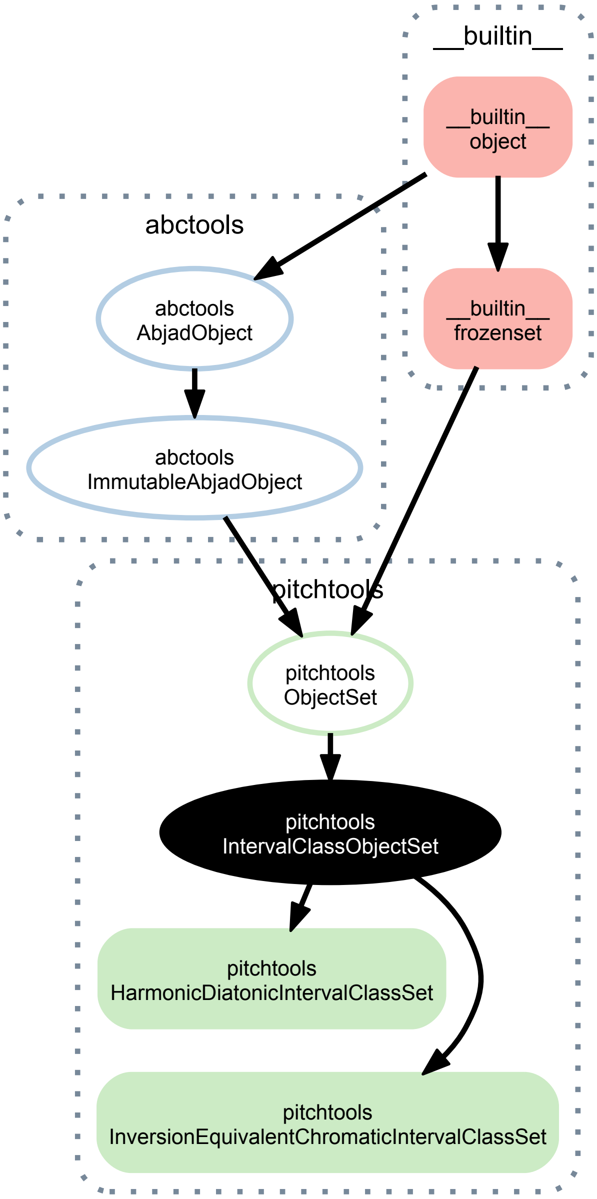
IntervalClassObjectSegment.**__ne__**()

x.__ne__(y) <==> x!=y

IntervalClassObjectSegment.**__repr__**()

IntervalClassObjectSegment.**__rmul__**(*n*)

21.1.17 pitchtools.IntervalClassObjectSet



class `pitchtools.`**`IntervalClassObjectSet`**(**args*, ***kwargs*)

New in version 2.0. Interval-class set base class.

Read-only properties

`IntervalClassObjectSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`IntervalClassObjectSet.`**`copy`**()

Return a shallow copy of a set.

`IntervalClassObjectSet.`**`difference`**()

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

IntervalClassObjectSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

IntervalClassObjectSet.**isdisjoint**()
Return True if two sets have a null intersection.

IntervalClassObjectSet.**issubset**()
Report whether another set contains this set.

IntervalClassObjectSet.**issuperset**()
Report whether this set contains another set.

IntervalClassObjectSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

IntervalClassObjectSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

IntervalClassObjectSet.**__and__**()
x.__and__(y) <==> x&y

IntervalClassObjectSet.**__cmp__**(*y) <==> cmp(x, y*)

IntervalClassObjectSet.**__contains__**()
x.__contains__(y) <==> y in x.

IntervalClassObjectSet.**__eq__**()
x.__eq__(y) <==> x==y

IntervalClassObjectSet.**__ge__**()
x.__ge__(y) <==> x>=y

IntervalClassObjectSet.**__gt__**()
x.__gt__(y) <==> x>y

IntervalClassObjectSet.**__hash__**(*) <==> hash(x*)

IntervalClassObjectSet.**__iter__**(*) <==> iter(x*)

IntervalClassObjectSet.**__le__**()
x.__le__(y) <==> x<=y

IntervalClassObjectSet.**__len__**(*) <==> len(x*)

IntervalClassObjectSet.**__lt__**()
x.__lt__(y) <==> x<y

IntervalClassObjectSet.**__ne__**()
x.__ne__(y) <==> x!=y

IntervalClassObjectSet.**__or__**()
x.__or__(y) <==> x|y

IntervalClassObjectSet.**__rand__**()
x.__rand__(y) <==> y&x

IntervalClassObjectSet.**__repr__**(*) <==> repr(x*)

IntervalClassObjectSet.**__ror__**()
x.__ror__(y) <==> y|x

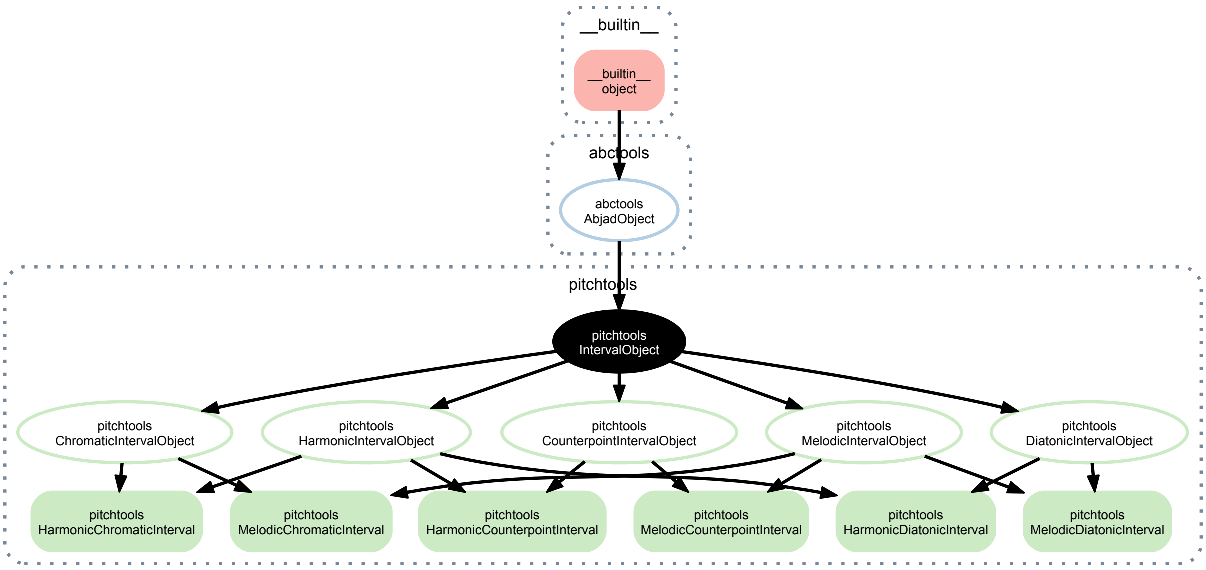
IntervalClassObjectSet.**__rsub__**()
x.__rsub__(y) <==> y-x

IntervalClassObjectSet.**__rxor__**()
x.__rxor__(y) <==> y^x

IntervalClassObjectSet.**__sub__**()
x.__sub__(y) <==> x-y

IntervalClassObjectSet.**__xor__**()
x.__xor__(y) <==> x^y

21.1.18 pitchtools.IntervalObject



class pitchtools.**IntervalObject**
New in version 2.0. Interval base class.

Read-only properties

IntervalObject.**cents**

IntervalObject.**interval_class**

IntervalObject.**number**

IntervalObject.**semitones**

IntervalObject.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

IntervalObject.**__abs__**()

IntervalObject.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

`IntervalObject.__float__()`

`IntervalObject.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`IntervalObject.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`IntervalObject.__hash__()`

`IntervalObject.__int__()`

`IntervalObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`IntervalObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

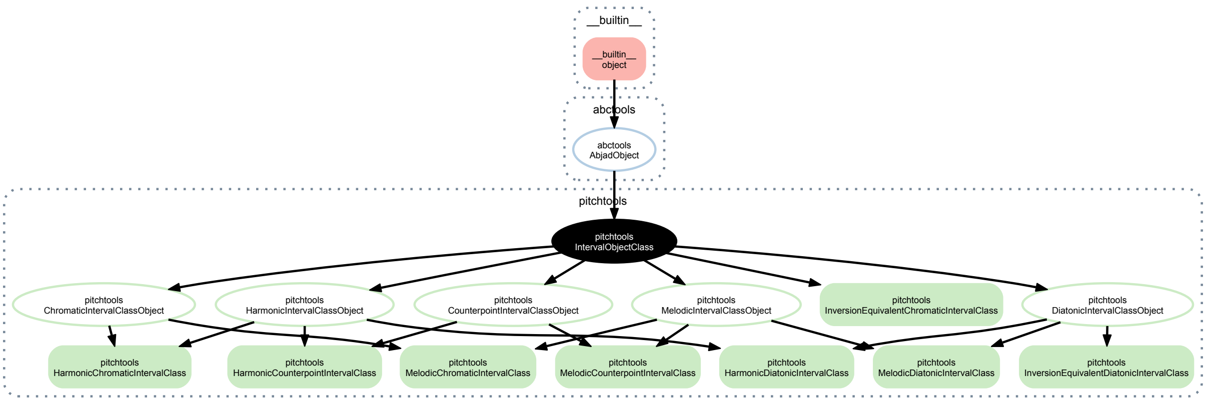
`IntervalObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`IntervalObject.__repr__()`

`IntervalObject.__str__()`

21.1.19 pitchtools.IntervalObjectClass



class `pitchtools.IntervalObjectClass`
New in version 2.0. Interval-class base class.

Read-only properties

`IntervalObjectClass.number`

`IntervalObjectClass.storage_format`
Storage format of Abjad object.

Return string.

Special methods

`IntervalObjectClass.__abs__()`

`IntervalObjectClass.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`IntervalObjectClass.__float__()`

`IntervalObjectClass.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`IntervalObjectClass.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`IntervalObjectClass.__hash__()`

`IntervalObjectClass.__int__()`

`IntervalObjectClass.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`IntervalObjectClass.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

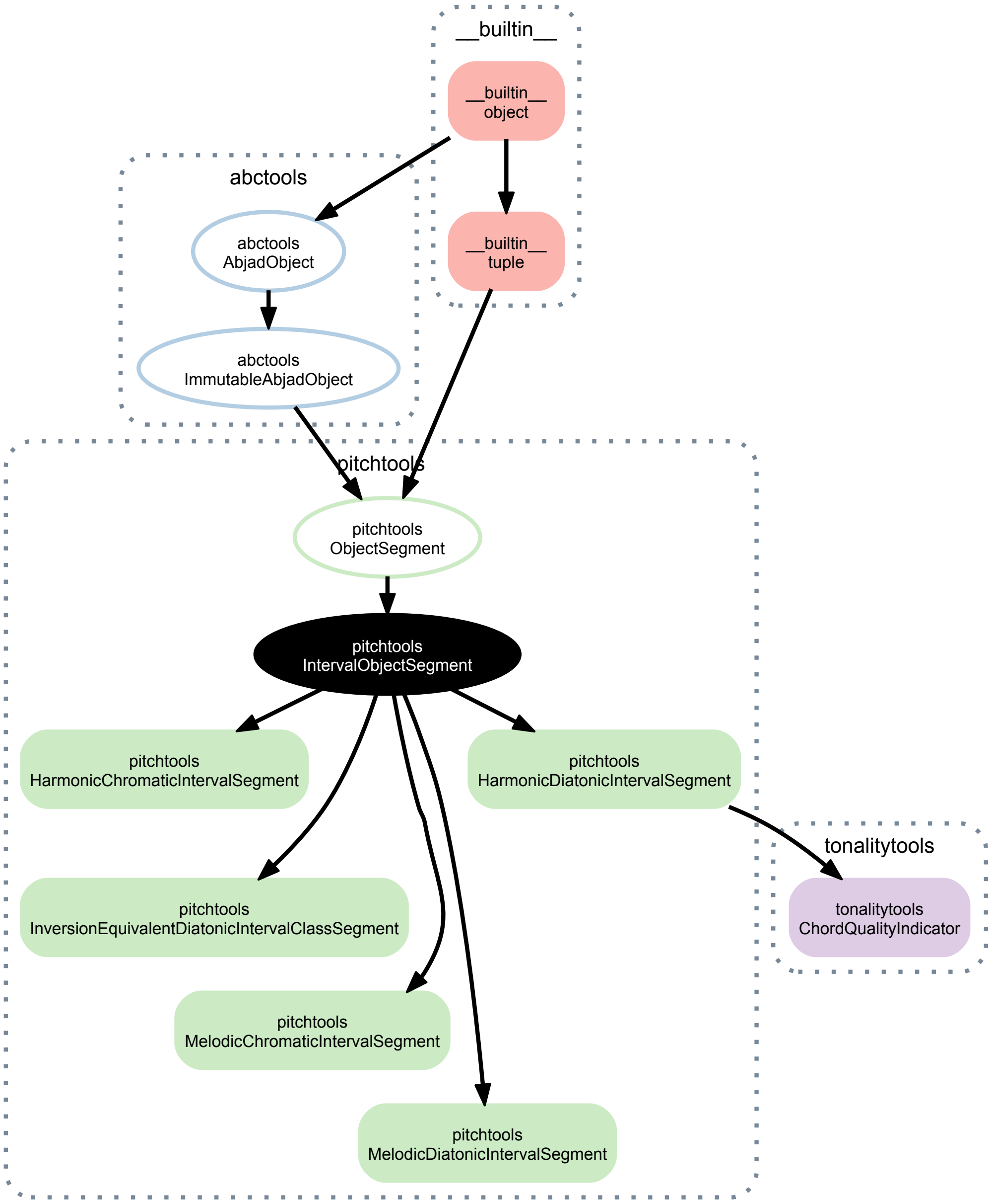
`IntervalObjectClass.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`IntervalObjectClass.__repr__()`

`IntervalObjectClass.__str__()`

21.1.20 pitchtools.IntervalObjectSegment



class `pitchtools.`**`IntervalObjectSegment`**(**args*, ***kwargs*)

New in version 2.0. Class of abstract ordered collection of interval instances from which concrete classes inherit.

Read-only properties

`IntervalObjectSegment.`**`interval_classes`**

`IntervalObjectSegment.`**`intervals`**

`IntervalObjectSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`IntervalObjectSegment.count` (*value*) → integer – return number of occurrences of value

`IntervalObjectSegment.index` (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`IntervalObjectSegment.rotate` (*n*)

Special methods

`IntervalObjectSegment.__add__` (*arg*)

`IntervalObjectSegment.__contains__` ()
x.__contains__(y) <==> y in x

`IntervalObjectSegment.__eq__` ()
x.__eq__(y) <==> x==y

`IntervalObjectSegment.__ge__` ()
x.__ge__(y) <==> x>=y

`IntervalObjectSegment.__getitem__` ()
x.__getitem__(y) <==> x[y]

`IntervalObjectSegment.__getslice__` (*start*, *stop*)

`IntervalObjectSegment.__gt__` ()
x.__gt__(y) <==> x>y

`IntervalObjectSegment.__hash__` () <==> *hash(x)*

`IntervalObjectSegment.__iter__` () <==> *iter(x)*

`IntervalObjectSegment.__le__` ()
x.__le__(y) <==> x<=y

`IntervalObjectSegment.__len__` () <==> *len(x)*

`IntervalObjectSegment.__lt__` ()
x.__lt__(y) <==> x<y

`IntervalObjectSegment.__mul__` (*n*)

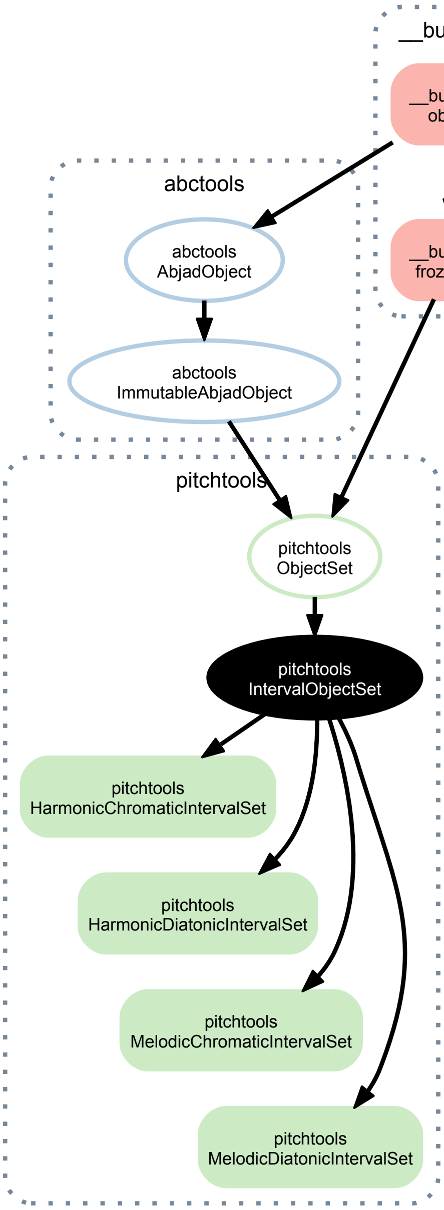
`IntervalObjectSegment.__ne__` ()
x.__ne__(y) <==> x!=y

`IntervalObjectSegment.__repr__` ()

`IntervalObjectSegment.__rmul__` (*n*)

`IntervalObjectSegment.__str__` ()

21.1.21 pitchtools.IntervalObjectSet



class `pitchtools.IntervalObjectSet` (**args*, ***kwargs*)

New in version 2.0. Abstract interval set.

Read-only properties

`IntervalObjectSet.storage_format`

Storage format of Abjad object.

Return string.

Methods

`IntervalObjectSet.copy()`

Return a shallow copy of a set.

`IntervalObjectSet.difference()`

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`IntervalObjectSet.`**intersection**`()`
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`IntervalObjectSet.`**isdisjoint**`()`
Return True if two sets have a null intersection.

`IntervalObjectSet.`**issubset**`()`
Report whether another set contains this set.

`IntervalObjectSet.`**issuperset**`()`
Report whether this set contains another set.

`IntervalObjectSet.`**symmetric_difference**`()`
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`IntervalObjectSet.`**union**`()`
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`IntervalObjectSet.`**__and__**`()`
x.__and__(y) <==> x&y

`IntervalObjectSet.`**__cmp__**`(`*y) <==> cmp(x, y*`)`

`IntervalObjectSet.`**__contains__**`()`
x.__contains__(y) <==> y in x.

`IntervalObjectSet.`**__eq__**`()`
x.__eq__(y) <==> x==y

`IntervalObjectSet.`**__ge__**`()`
x.__ge__(y) <==> x>=y

`IntervalObjectSet.`**__gt__**`()`
x.__gt__(y) <==> x>y

`IntervalObjectSet.`**__hash__**`(`*) <==> hash(x*`)`

`IntervalObjectSet.`**__iter__**`(`*) <==> iter(x*`)`

`IntervalObjectSet.`**__le__**`()`
x.__le__(y) <==> x<=y

`IntervalObjectSet.`**__len__**`(`*) <==> len(x*`)`

`IntervalObjectSet.`**__lt__**`()`
x.__lt__(y) <==> x<y

`IntervalObjectSet.`**__ne__**`()`
x.__ne__(y) <==> x!=y

`IntervalObjectSet.`**__or__**`()`
x.__or__(y) <==> x|y

`IntervalObjectSet.`**__rand__**`()`
x.__rand__(y) <==> y&x

`IntervalObjectSet.`**__repr__**`(`*) <==> repr(x*`)`

`IntervalObjectSet.__ror__()`
x.__ror__(y) <==> y|x

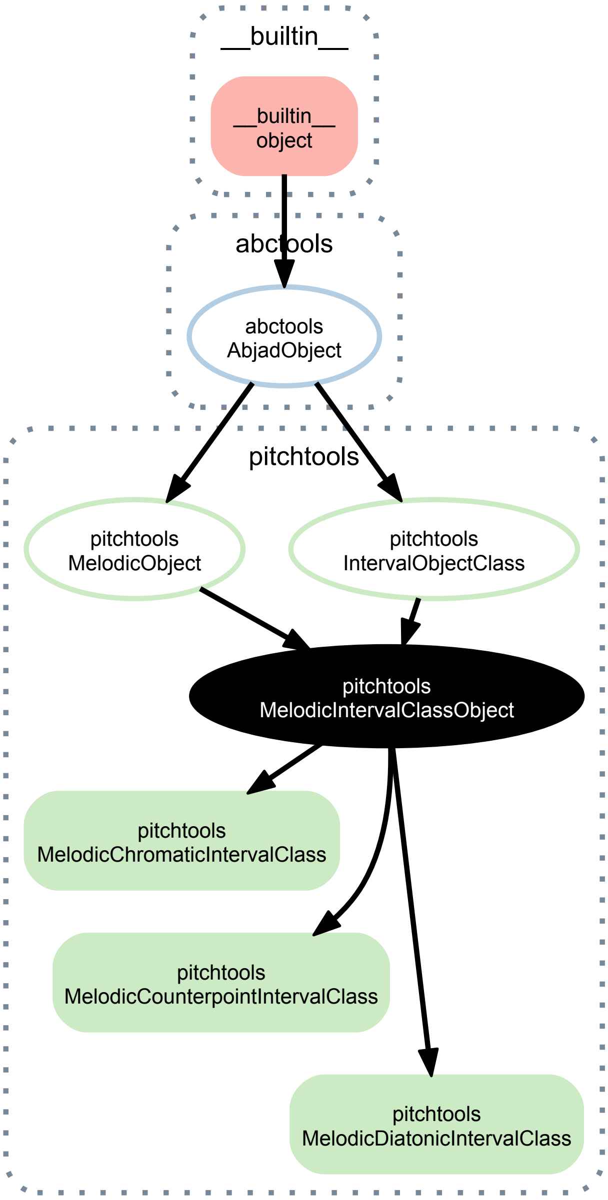
`IntervalObjectSet.__rsub__()`
x.__rsub__(y) <==> y-x

`IntervalObjectSet.__rxor__()`
x.__rxor__(y) <==> y^x

`IntervalObjectSet.__sub__()`
x.__sub__(y) <==> x-y

`IntervalObjectSet.__xor__()`
x.__xor__(y) <==> x^y

21.1.22 pitchtools.MelodicIntervalClassObject



class `pitchtools.MelodicIntervalClassObject`
New in version 2.0. Melodic interval-class base class.

Read-only properties

`MelodicIntervalClassObject.`**`direction_number`**

`MelodicIntervalClassObject.`**`direction_symbol`**

`MelodicIntervalClassObject.`**`direction_word`**

`MelodicIntervalClassObject.`**`number`**

`MelodicIntervalClassObject.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`MelodicIntervalClassObject.`**`__abs__`**`()`

`MelodicIntervalClassObject.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`MelodicIntervalClassObject.`**`__float__`**`()`

`MelodicIntervalClassObject.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MelodicIntervalClassObject.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`MelodicIntervalClassObject.`**`__hash__`**`()`

`MelodicIntervalClassObject.`**`__int__`**`()`

`MelodicIntervalClassObject.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MelodicIntervalClassObject.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

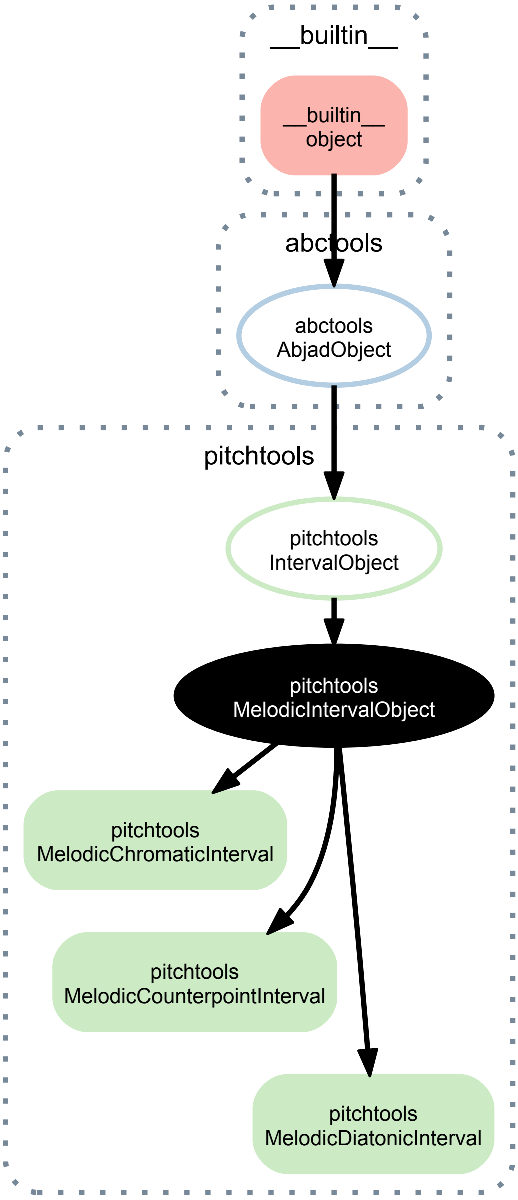
`MelodicIntervalClassObject.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`MelodicIntervalClassObject.`**`__repr__`**`()`

`MelodicIntervalClassObject.`**`__str__`**`()`

21.1.23 pitchtools.MelodicIntervalObject



class `pitchtools.`**`MelodicIntervalObject`**

New in version 2.0. Melodic interval base class.

Read-only properties

`MelodicIntervalObject.`**`cents`**

`MelodicIntervalObject.`**`direction_number`**

`MelodicIntervalObject.`**`direction_string`**

`MelodicIntervalObject.`**`interval_class`**

`MelodicIntervalObject.`**`number`**

`MelodicIntervalObject.`**`semitones`**

`MelodicIntervalObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

MelodicIntervalObject.**__abs__**()

MelodicIntervalObject.**__eq__**(*arg*)

MelodicIntervalObject.**__float__**()

MelodicIntervalObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicIntervalObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MelodicIntervalObject.**__hash__**()

MelodicIntervalObject.**__int__**()

MelodicIntervalObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicIntervalObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

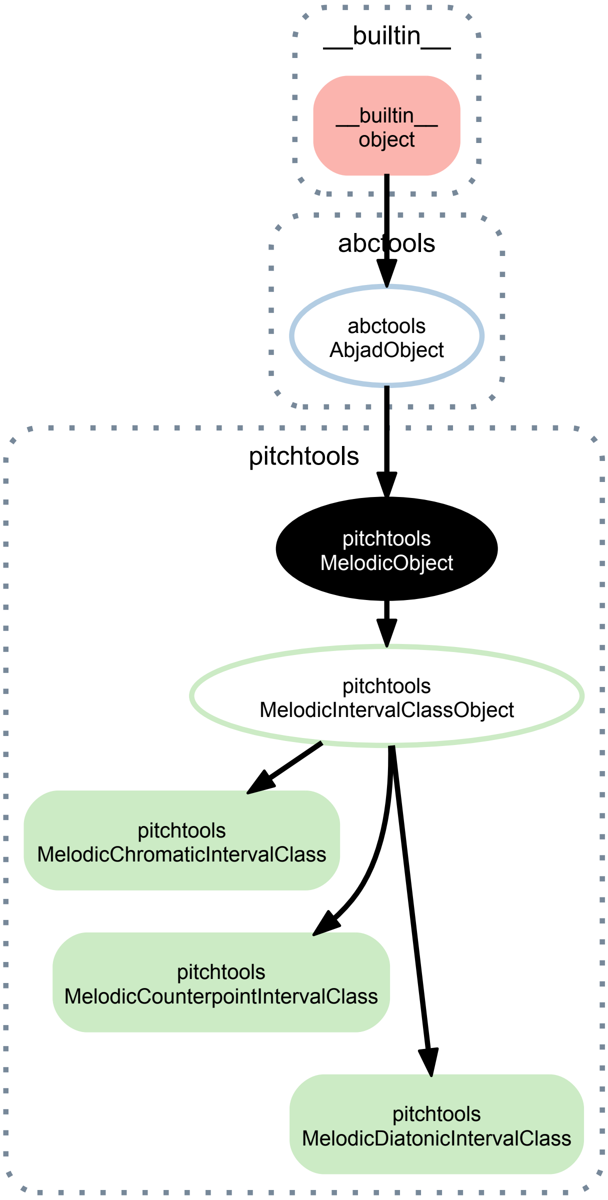
MelodicIntervalObject.**__ne__**(*arg*)

MelodicIntervalObject.**__neg__**()

MelodicIntervalObject.**__repr__**()

MelodicIntervalObject.**__str__**()

21.1.24 pitchtools.MelodicObject



class `pitchtools.`**`MelodicObject`**

..versionadded:: 2.0

Melodic object base class.

Read-only properties

`MelodicObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`MelodicObject.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`MelodicObject.__ge__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`MelodicObject.__gt__(expr)`

Abjad objects by default do not implement this method.

Raise exception

`MelodicObject.__le__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`MelodicObject.__lt__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`MelodicObject.__ne__(expr)`

Defined equal to the opposite of equality.

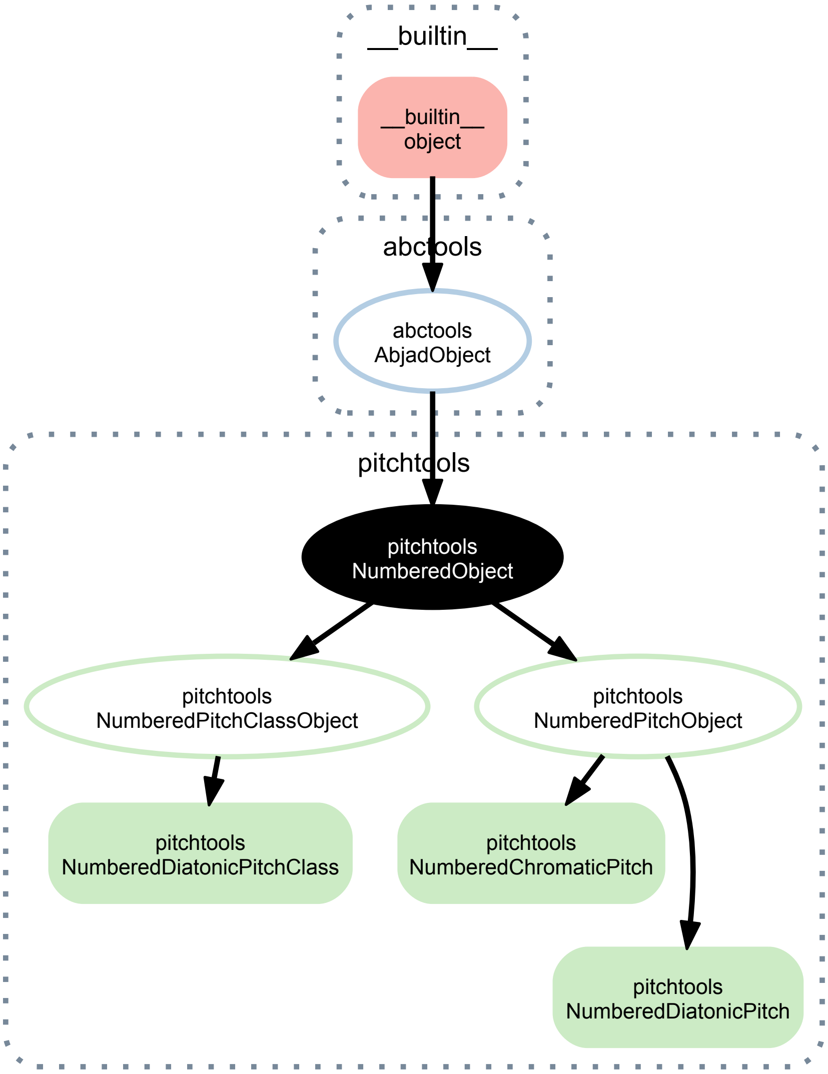
Return boolean.

`MelodicObject.__repr__()`

Interpreter representation of Abjad object.

Return string.

21.1.25 pitchtools.NumberedObject



class `pitchtools.`**`NumberedObject`**

..versionadded: 1.1.2

Numbered object base class.

Read-only properties

`NumberedObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`NumberedObject.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`NumberedObject.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`NumberedObject.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`NumberedObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NumberedObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

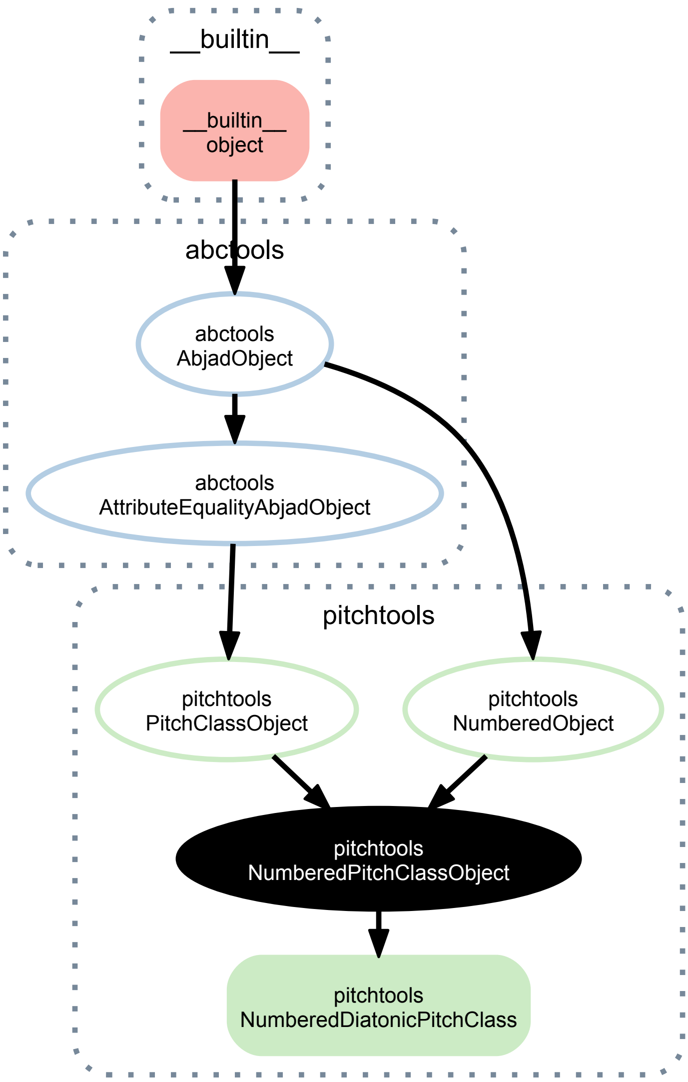
`NumberedObject.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`NumberedObject.__repr__()`
Interpreter representation of Abjad object.

Return string.

21.1.26 pitchtools.NumberedPitchClassObject



class pitchtools.**NumberedPitchClassObject**
New in version 2.0. Numbered pitch-class base class.

Read-only properties

NumberedPitchClassObject.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

NumberedPitchClassObject.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchClassObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NumberedPitchClassObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

NumberedPitchClassObject.**__hash__**()

NumberedPitchClassObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NumberedPitchClassObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

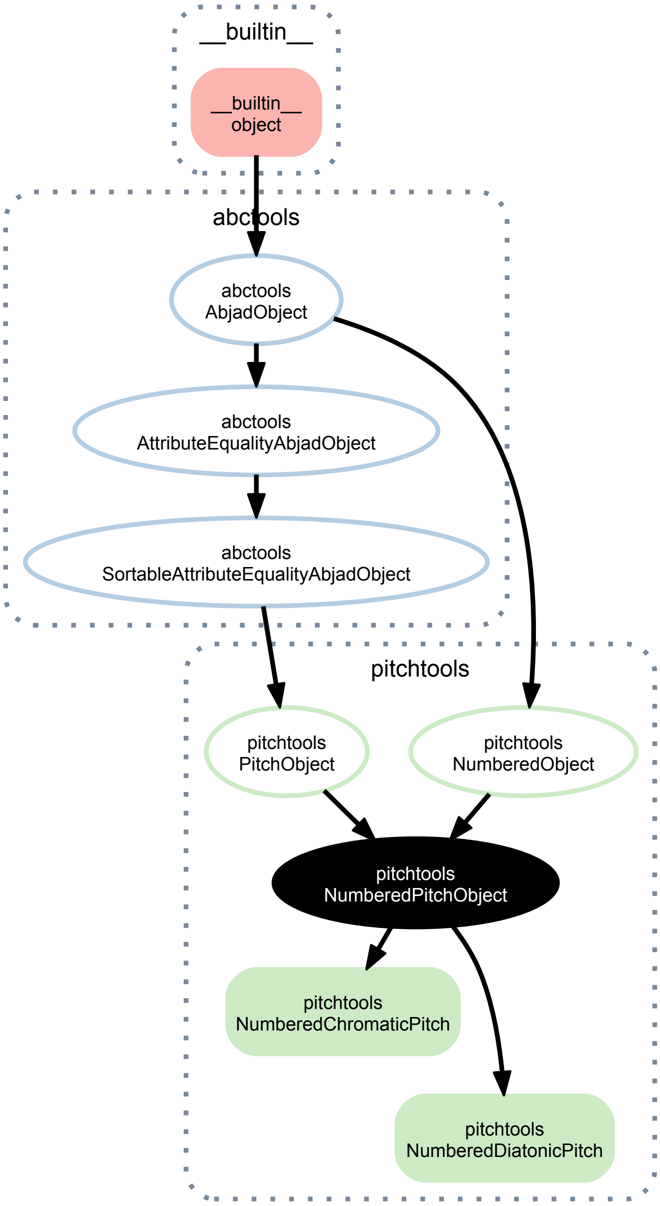
NumberedPitchClassObject.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchClassObject.**__repr__**()
Interpreter representation of object defined equal to class name and format string.

Return string.

21.1.27 pitchtools.NumberedPitchObject



class `pitchtools.`**`NumberedPitchObject`**

New in version 2.0. Numbered pitch base class from which concrete classes inherit.

Read-only properties

`NumberedPitchObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`NumberedPitchObject.`**`__abs__`**`()`

`NumberedPitchObject.`**`__eq__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchObject.**__float__**()

NumberedPitchObject.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchObject.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchObject.**__hash__**()

NumberedPitchObject.**__int__**()

NumberedPitchObject.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

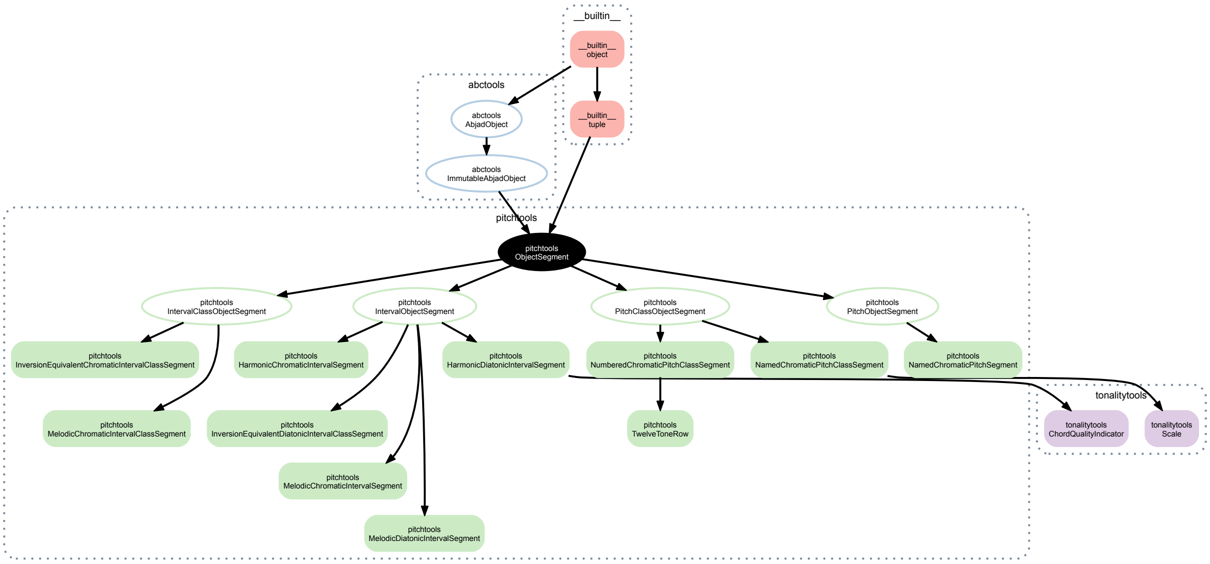
NumberedPitchObject.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedPitchObject.**__ne__**(*arg*)

NumberedPitchObject.**__repr__**()

21.1.28 pitchtools.ObjectSegment



class pitchtools.**ObjectSegment**(**args*, ***kwargs*)
New in version 2.0. Mix-in base class for ordered collections of pitch objects.

Read-only properties

ObjectSegment.**storage_format**
Storage format of Abjad object.

Return string.

Methods

`ObjectSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`ObjectSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`ObjectSegment.`**`__add__`**(*arg*)

`ObjectSegment.`**`__contains__`**()
x.__contains__(y) <==> y in x

`ObjectSegment.`**`__eq__`**()
x.__eq__(y) <==> x==y

`ObjectSegment.`**`__ge__`**()
x.__ge__(y) <==> x>=y

`ObjectSegment.`**`__getitem__`**()
x.__getitem__(y) <==> x[y]

`ObjectSegment.`**`__getslice__`**(*start*, *stop*)

`ObjectSegment.`**`__gt__`**()
x.__gt__(y) <==> x>y

`ObjectSegment.`**`__hash__`**() <==> *hash(x)*

`ObjectSegment.`**`__iter__`**() <==> *iter(x)*

`ObjectSegment.`**`__le__`**()
x.__le__(y) <==> x<=y

`ObjectSegment.`**`__len__`**() <==> *len(x)*

`ObjectSegment.`**`__lt__`**()
x.__lt__(y) <==> x<y

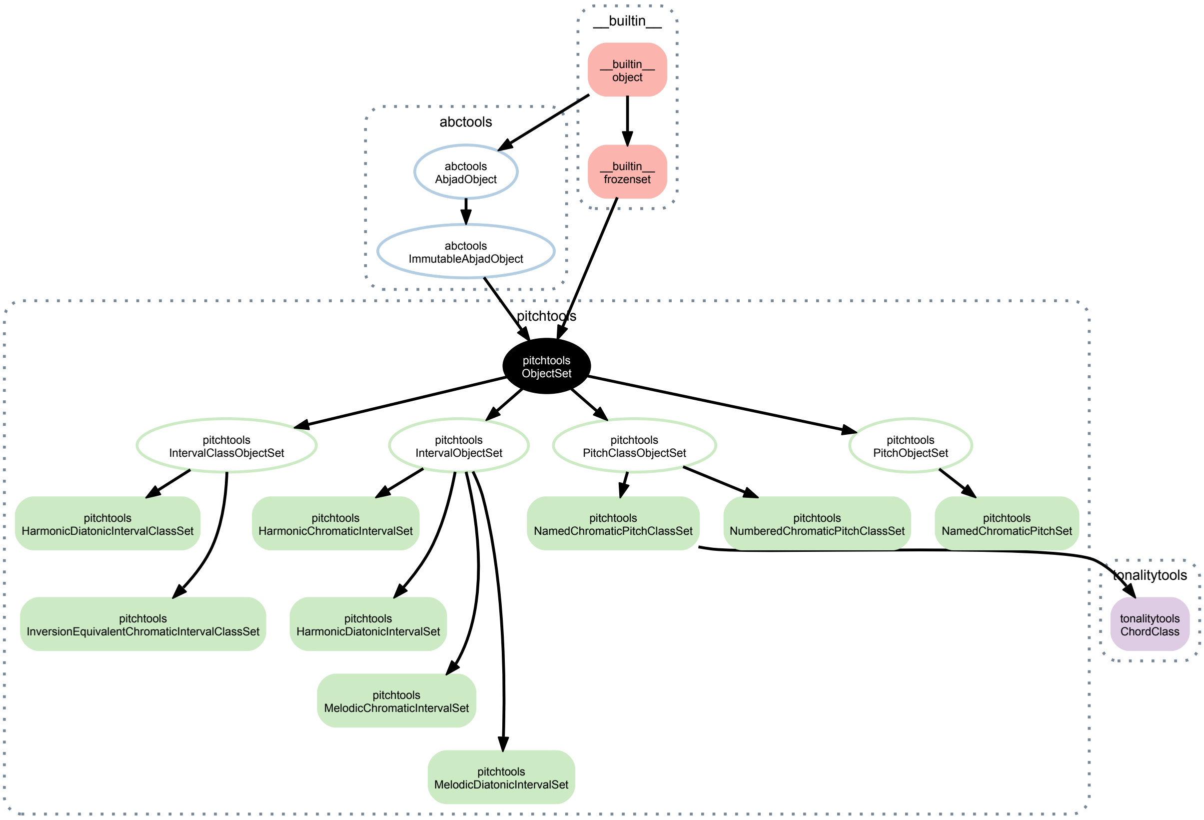
`ObjectSegment.`**`__mul__`**(*n*)

`ObjectSegment.`**`__ne__`**()
x.__ne__(y) <==> x!=y

`ObjectSegment.`**`__repr__`**() <==> *repr(x)*

`ObjectSegment.`**`__rmul__`**(*n*)

21.1.29 pitchtools.ObjectSet



class `pitchtools.ObjectSet` (**args*, ***kwargs*)
New in version 2.0. Music-theoretic set base class.

Read-only properties

`ObjectSet.storage_format`
Storage format of Abjad object.

Return string.

Methods

`ObjectSet.copy()`
Return a shallow copy of a set.

`ObjectSet.difference()`
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`ObjectSet.intersection()`
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`ObjectSet.isdisjoint()`
Return True if two sets have a null intersection.

`ObjectSet.issubset()`
Report whether another set contains this set.

`ObjectSet.issuperset()`
Report whether this set contains another set.

ObjectSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

ObjectSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

ObjectSet.**__and__**()
x.__and__(y) <==> x&y

ObjectSet.**__cmp__**(*y*) <==> *cmp(x, y)*

ObjectSet.**__contains__**()
x.__contains__(y) <==> y in x.

ObjectSet.**__eq__**()
x.__eq__(y) <==> x==y

ObjectSet.**__ge__**()
x.__ge__(y) <==> x>=y

ObjectSet.**__gt__**()
x.__gt__(y) <==> x>y

ObjectSet.**__hash__**() <==> *hash(x)*

ObjectSet.**__iter__**() <==> *iter(x)*

ObjectSet.**__le__**()
x.__le__(y) <==> x<=y

ObjectSet.**__len__**() <==> *len(x)*

ObjectSet.**__lt__**()
x.__lt__(y) <==> x<y

ObjectSet.**__ne__**()
x.__ne__(y) <==> x!=y

ObjectSet.**__or__**()
x.__or__(y) <==> x|y

ObjectSet.**__rand__**()
x.__rand__(y) <==> y&x

ObjectSet.**__repr__**() <==> *repr(x)*

ObjectSet.**__ror__**()
x.__ror__(y) <==> y|x

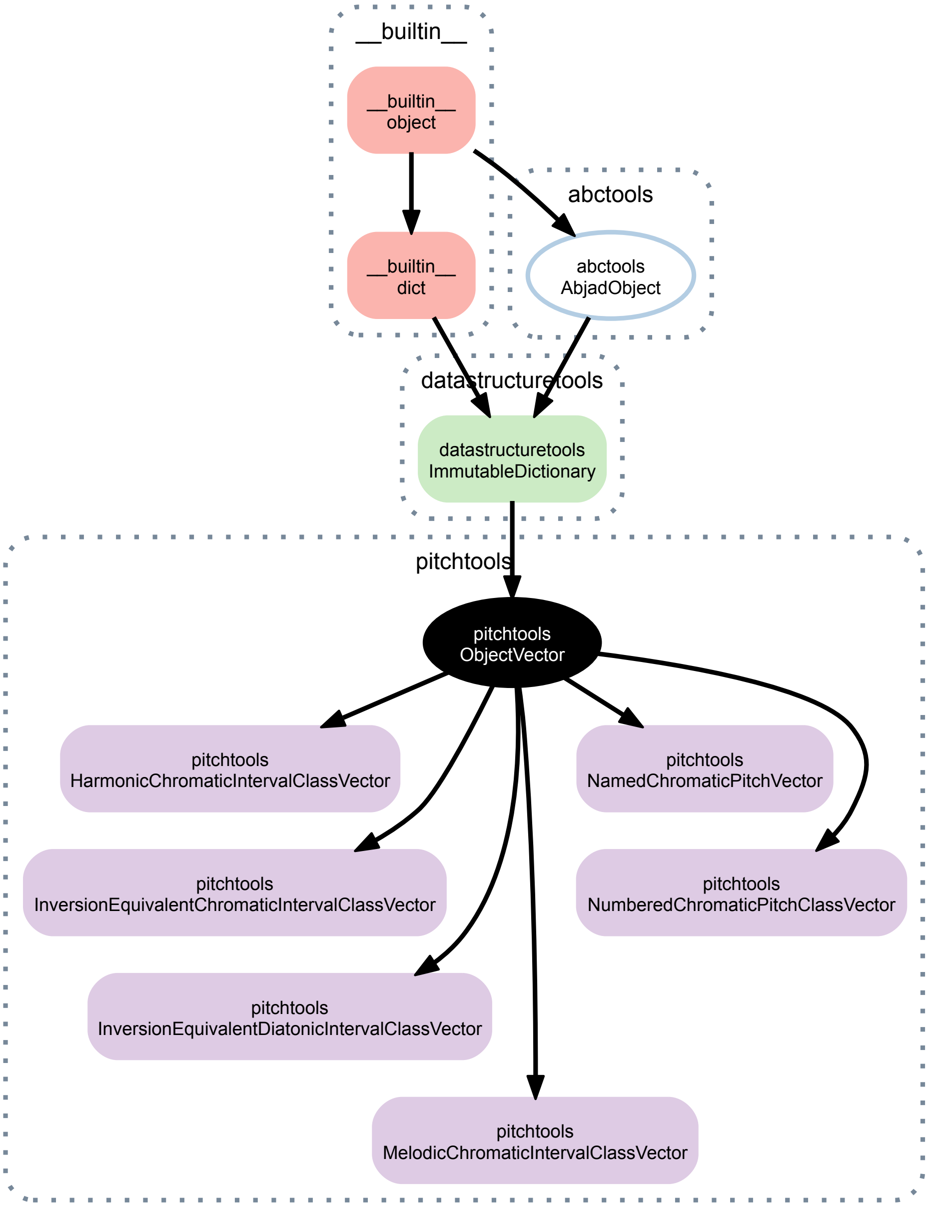
ObjectSet.**__rsub__**()
x.__rsub__(y) <==> y-x

ObjectSet.**__rxor__**()
x.__rxor__(y) <==> y^x

ObjectSet.**__sub__**()
x.__sub__(y) <==> x-y

ObjectSet.**__xor__**()
x.__xor__(y) <==> x^y

21.1.30 pitchtools.ObjectVector



class `pitchtools.ObjectVector`

New in version 2.0. Music theoretic vector base class.

Read-only properties

`ObjectVector.storage_format`

Storage format of Abjad object.

Return string.

Methods

`ObjectVector.clear()` → None. Remove all items from D.

`ObjectVector.copy()` → a shallow copy of D

`ObjectVector.get(k[, d])` → D[k] if k in D, else d. d defaults to None.

`ObjectVector.`**`has_key`**`(k)` → True if D has a key k, else False

`ObjectVector.`**`items`**`()` → list of D's (key, value) pairs, as 2-tuples

`ObjectVector.`**`iteritems`**`()` → an iterator over the (key, value) items of D

`ObjectVector.`**`iterkeys`**`()` → an iterator over the keys of D

`ObjectVector.`**`itervalues`**`()` → an iterator over the values of D

`ObjectVector.`**`keys`**`()` → list of D's keys

`ObjectVector.`**`pop`**`(k[, d])` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`ObjectVector.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`ObjectVector.`**`setdefault`**`(k[, d])` → D.get(k,d), also set D[k]=d if k not in D

`ObjectVector.`**`update`**`([E], **F)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`ObjectVector.`**`values`**`()` → list of D's values

`ObjectVector.`**`viewitems`**`()` → a set-like object providing a view on D's items

`ObjectVector.`**`viewkeys`**`()` → a set-like object providing a view on D's keys

`ObjectVector.`**`viewvalues`**`()` → an object providing a view on D's values

Special methods

`ObjectVector.`**`__cmp__`**`(y)` <==> *cmp(x, y)*

`ObjectVector.`**`__contains__`**`(k)` → True if D has a key k, else False

`ObjectVector.`**`__delitem__`**`(*args)`

`ObjectVector.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`ObjectVector.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`ObjectVector.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`ObjectVector.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`ObjectVector.`**`__iter__`**`()` <==> *iter(x)*

`ObjectVector.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`ObjectVector.`**`__len__`**`()` <==> *len(x)*

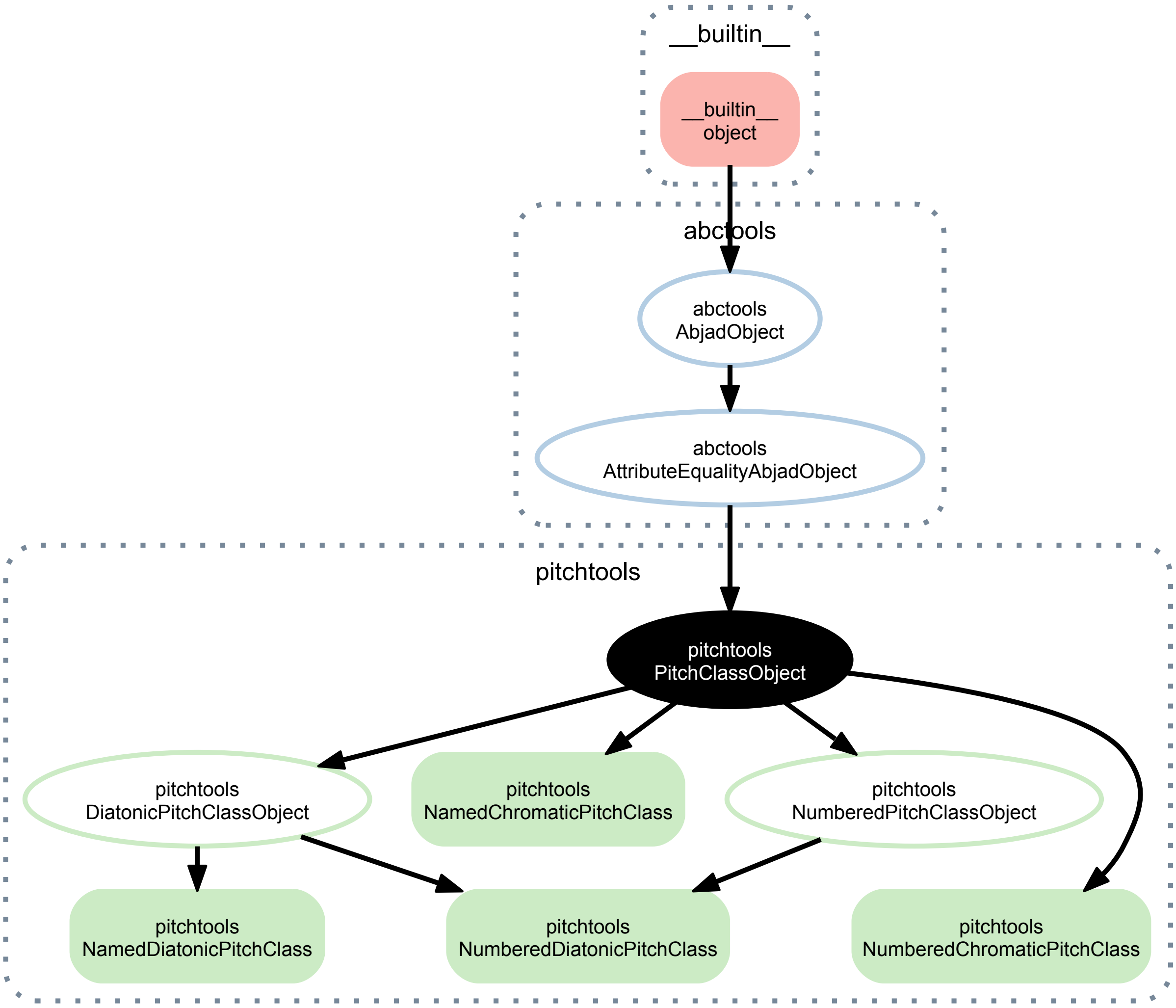
`ObjectVector.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`ObjectVector.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`ObjectVector.`**`__repr__`**`()` <==> *repr(x)*

`ObjectVector.`**`__setitem__`**`(*args)`

21.1.31 pitchtools.PitchClassObject



class `pitchtools.`**`PitchClassObject`**

New in version 2.0. Pitch-class base class.

Read-only properties

`PitchClassObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`PitchClassObject.`**`__eq__`**(*arg*)

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`PitchClassObject.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`PitchClassObject.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`PitchClassObject.__hash__()`

`PitchClassObject.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PitchClassObject.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

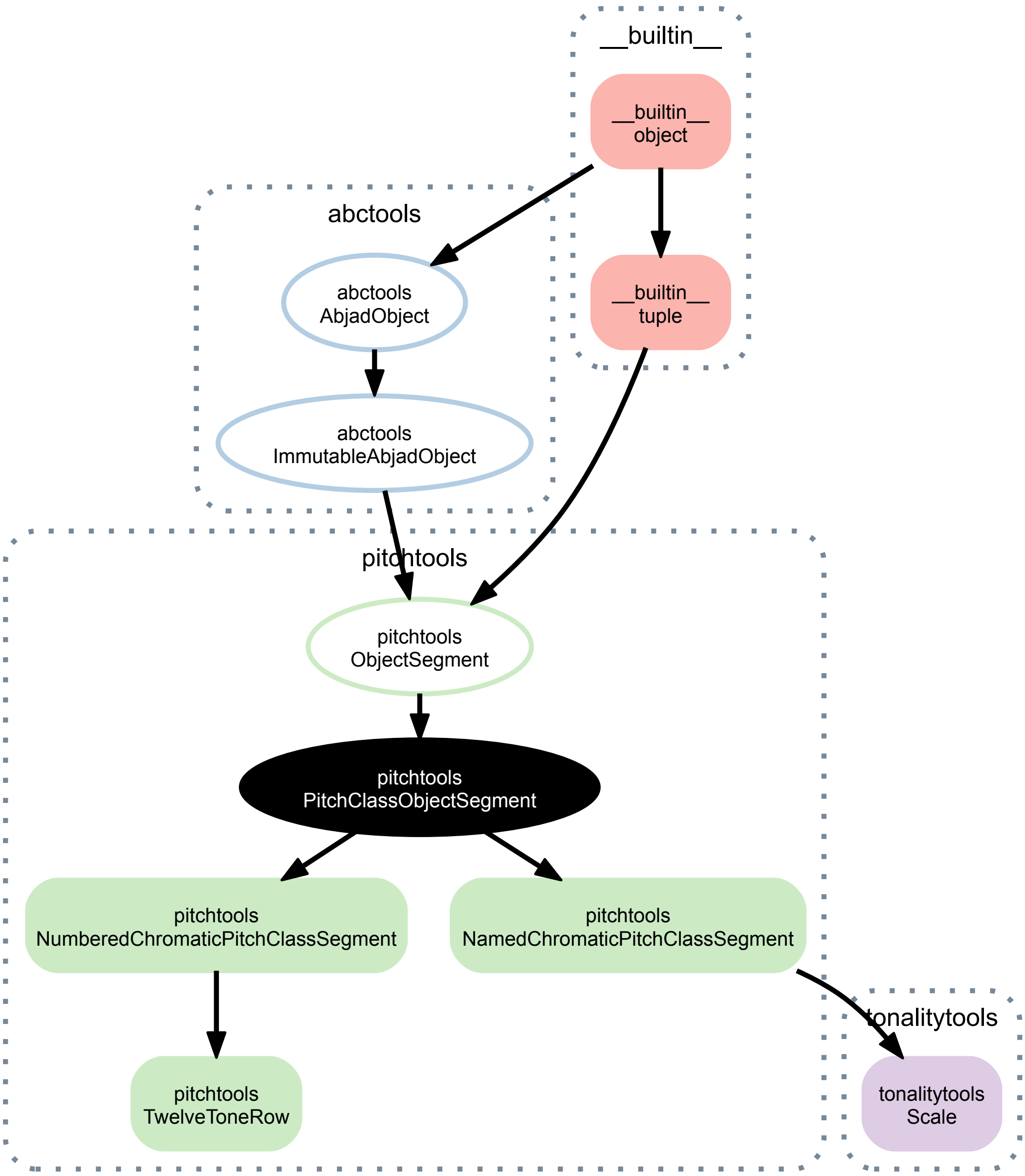
`PitchClassObject.__ne__(arg)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`PitchClassObject.__repr__()`
Interpreter representation of object defined equal to class name and format string.

Return string.

21.1.32 pitchtools.PitchClassObjectSegment



class `pitchtools.`**`PitchClassObjectSegment`**(**args*, ***kwargs*)

New in version 2.0. Pitch-class segment base class.

Read-only properties

`PitchClassObjectSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`PitchClassObjectSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`PitchClassObjectSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Special methods

PitchClassObjectSegment.**__add__**(*arg*)

PitchClassObjectSegment.**__contains__**()
x.__contains__(y) <==> y in x

PitchClassObjectSegment.**__eq__**()
x.__eq__(y) <==> x==y

PitchClassObjectSegment.**__ge__**()
x.__ge__(y) <==> x>=y

PitchClassObjectSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

PitchClassObjectSegment.**__getslice__**(*start*, *stop*)

PitchClassObjectSegment.**__gt__**()
x.__gt__(y) <==> x>y

PitchClassObjectSegment.**__hash__**() <==> *hash*(*x*)

PitchClassObjectSegment.**__iter__**() <==> *iter*(*x*)

PitchClassObjectSegment.**__le__**()
x.__le__(y) <==> x<=y

PitchClassObjectSegment.**__len__**() <==> *len*(*x*)

PitchClassObjectSegment.**__lt__**()
x.__lt__(y) <==> x<y

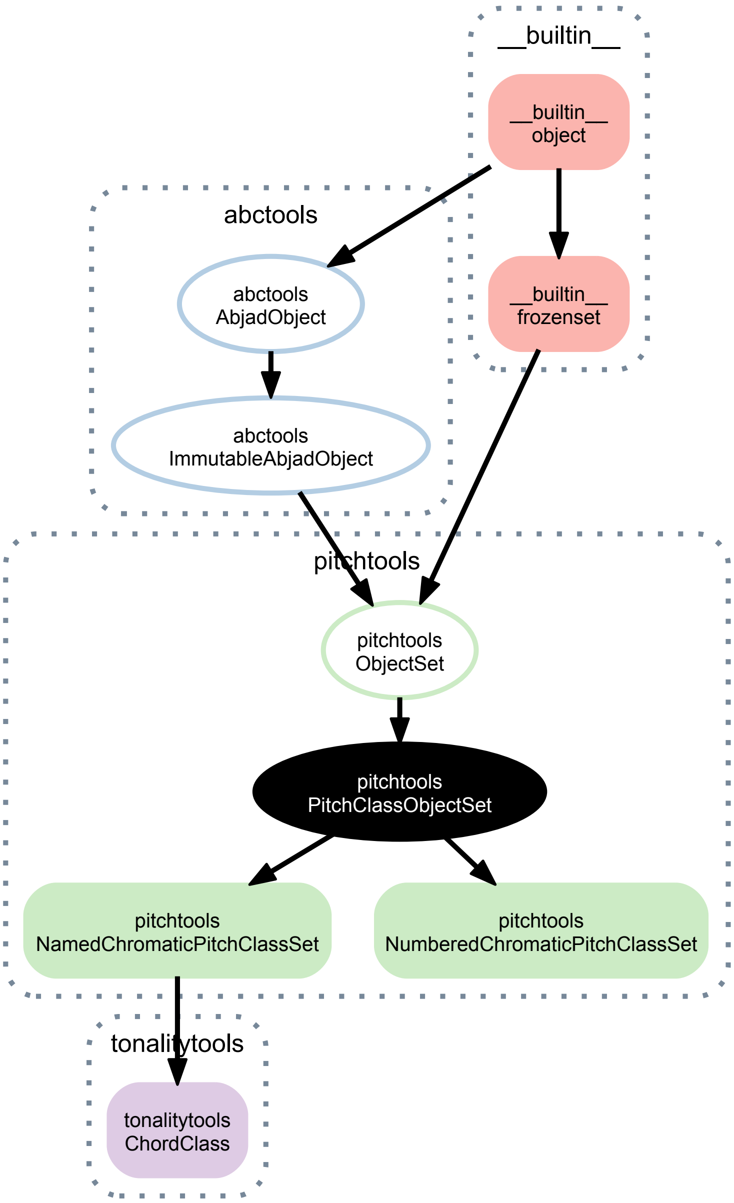
PitchClassObjectSegment.**__mul__**(*n*)

PitchClassObjectSegment.**__ne__**()
x.__ne__(y) <==> x!=y

PitchClassObjectSegment.**__repr__**() <==> *repr*(*x*)

PitchClassObjectSegment.**__rmul__**(*n*)

21.1.33 pitchtools.PitchClassObjectSet



class `pitchtools.`**`PitchClassObjectSet`**(**args*, ***kwargs*)
New in version 2.0. Pitch-class set base class.

Read-only properties

`PitchClassObjectSet.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`PitchClassObjectSet.`**`copy`**()
Return a shallow copy of a set.

`PitchClassObjectSet.`**`difference`**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`PitchClassObjectSet.`**`intersection`**`()`
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`PitchClassObjectSet.`**`isdisjoint`**`()`
Return True if two sets have a null intersection.

`PitchClassObjectSet.`**`issubset`**`()`
Report whether another set contains this set.

`PitchClassObjectSet.`**`issuperset`**`()`
Report whether this set contains another set.

`PitchClassObjectSet.`**`symmetric_difference`**`()`
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`PitchClassObjectSet.`**`union`**`()`
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`PitchClassObjectSet.`**`__and__`**`()`
x.__and__(y) <==> x&y

`PitchClassObjectSet.`**`__cmp__`**`(y)` <==> *cmp(x, y)*

`PitchClassObjectSet.`**`__contains__`**`()`
x.__contains__(y) <==> y in x.

`PitchClassObjectSet.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`PitchClassObjectSet.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`PitchClassObjectSet.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`PitchClassObjectSet.`**`__hash__`**`()` <==> *hash(x)*

`PitchClassObjectSet.`**`__iter__`**`()` <==> *iter(x)*

`PitchClassObjectSet.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`PitchClassObjectSet.`**`__len__`**`()` <==> *len(x)*

`PitchClassObjectSet.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`PitchClassObjectSet.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`PitchClassObjectSet.`**`__or__`**`()`
x.__or__(y) <==> x|y

`PitchClassObjectSet.`**`__rand__`**`()`
x.__rand__(y) <==> y&x

`PitchClassObjectSet.`**`__repr__`**`()` <==> *repr(x)*

`PitchClassObjectSet.`**`__ror__`**`()`
x.__ror__(y) <==> y|x

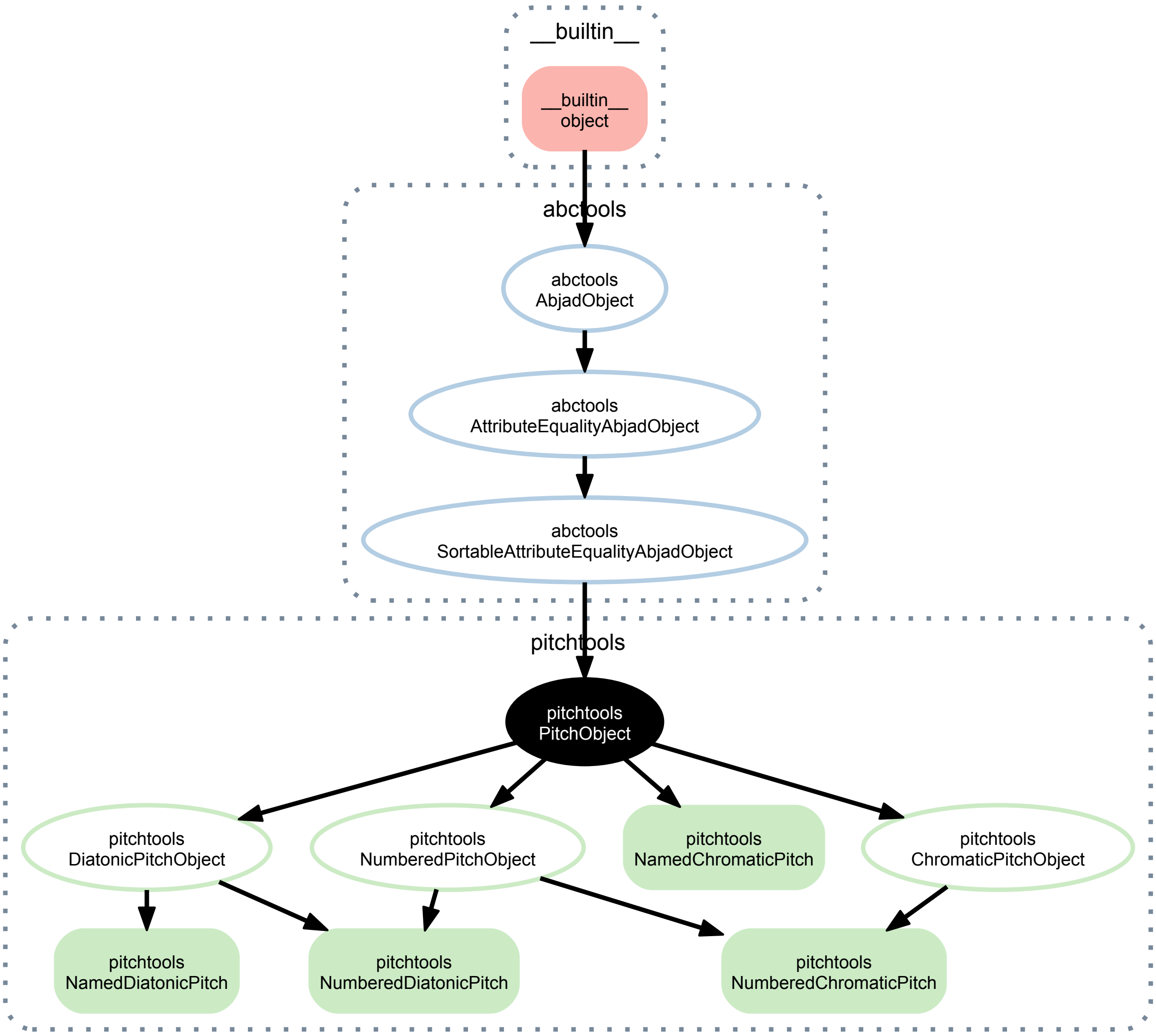
`PitchClassObjectSet.`**`__rsub__`**`()`
x.__rsub__(y) <==> y-x

`PitchClassObjectSet.`**`__rxor__`**`()`
x.__rxor__(y) <==> y^x

`PitchClassObjectSet.`**`__sub__`**`()`
x.__sub__(y) <==> x-y

`PitchClassObjectSet.`**`__xor__`**`()`
x.__xor__(y) <==> x^y

21.1.34 pitchtools.PitchObject



class `pitchtools.`**`PitchObject`**
New in version 2.0. Pitch base class.

Read-only properties

`PitchObject.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

PitchObject.**__abs__**()

PitchObject.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

PitchObject.**__float__**()

PitchObject.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

PitchObject.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

PitchObject.**__hash__**()

PitchObject.**__int__**()

PitchObject.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

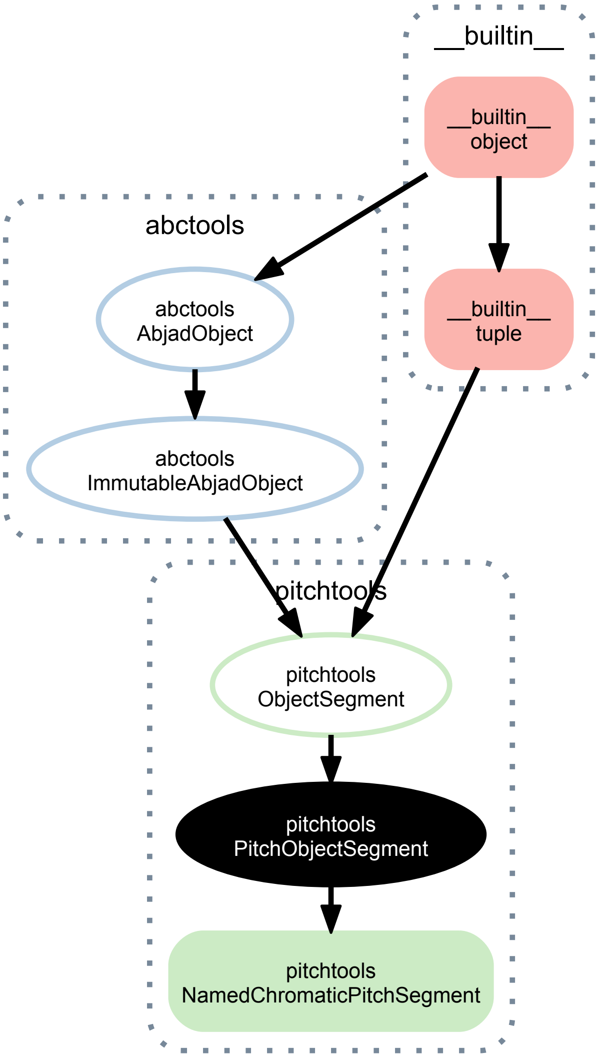
PitchObject.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

PitchObject.**__ne__**(*arg*)

PitchObject.**__repr__**()

21.1.35 pitchtools.PitchObjectSegment



class `pitchtools.`**`PitchObjectSegment`**(**args*, ***kwargs*)
New in version 2.0. Pitch segment base class.

Read-only properties

`PitchObjectSegment.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`PitchObjectSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`PitchObjectSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`PitchObjectSegment.`**`__add__`**(*arg*)

PitchObjectSegment.**__contains__**()
x.__contains__(y) <==> y in x

PitchObjectSegment.**__eq__**()
x.__eq__(y) <==> x==y

PitchObjectSegment.**__ge__**()
x.__ge__(y) <==> x>=y

PitchObjectSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

PitchObjectSegment.**__getslice__**(*start*, *stop*)

PitchObjectSegment.**__gt__**()
x.__gt__(y) <==> x>y

PitchObjectSegment.**__hash__**() <==> *hash(x)*

PitchObjectSegment.**__iter__**() <==> *iter(x)*

PitchObjectSegment.**__le__**()
x.__le__(y) <==> x<=y

PitchObjectSegment.**__len__**() <==> *len(x)*

PitchObjectSegment.**__lt__**()
x.__lt__(y) <==> x<y

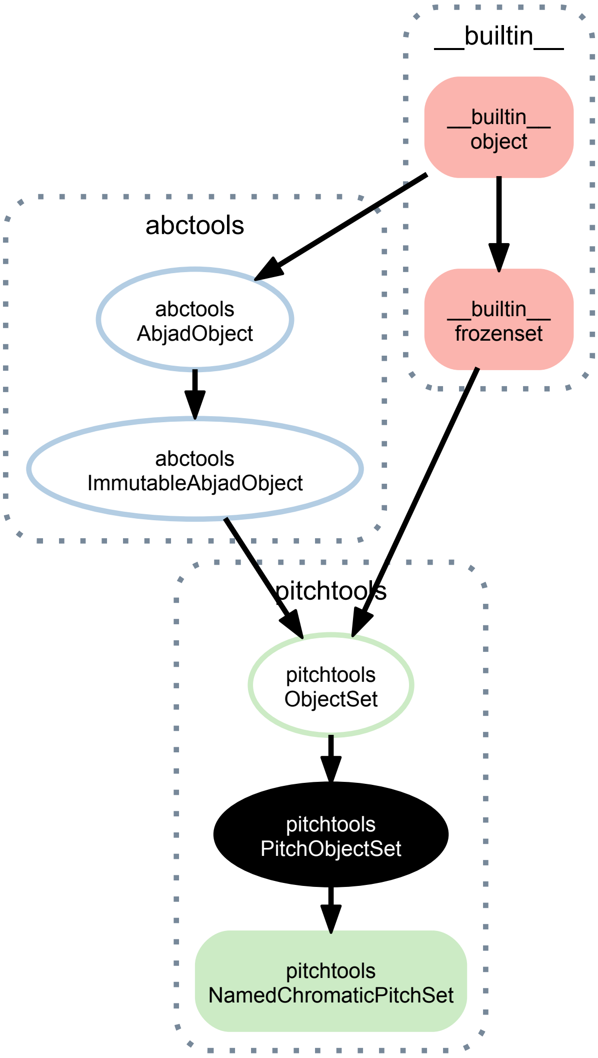
PitchObjectSegment.**__mul__**(*n*)

PitchObjectSegment.**__ne__**()
x.__ne__(y) <==> x!=y

PitchObjectSegment.**__repr__**() <==> *repr(x)*

PitchObjectSegment.**__rmul__**(*n*)

21.1.36 pitchtools.PitchObjectSet



class `pitchtools.`**`PitchObjectSet`**(**args*, ***kwargs*)

New in version 2.0. Pitch set base class.

Read-only properties

`PitchObjectSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`PitchObjectSet.`**`copy`**()

Return a shallow copy of a set.

`PitchObjectSet.`**`difference`**()

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`PitchObjectSet.`**`intersection`**()

Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`PitchObjectSet.isdisjoint()`
Return True if two sets have a null intersection.

`PitchObjectSet.issubset()`
Report whether another set contains this set.

`PitchObjectSet.issuperset()`
Report whether this set contains another set.

`PitchObjectSet.symmetric_difference()`
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`PitchObjectSet.union()`
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`PitchObjectSet.__and__()`
x.__and__(y) <==> x&y

`PitchObjectSet.__cmp__(y)` <==> *cmp(x, y)*

`PitchObjectSet.__contains__()`
x.__contains__(y) <==> y in x.

`PitchObjectSet.__eq__()`
x.__eq__(y) <==> x==y

`PitchObjectSet.__ge__()`
x.__ge__(y) <==> x>=y

`PitchObjectSet.__gt__()`
x.__gt__(y) <==> x>y

`PitchObjectSet.__hash__()` <==> *hash(x)*

`PitchObjectSet.__iter__()` <==> *iter(x)*

`PitchObjectSet.__le__()`
x.__le__(y) <==> x<=y

`PitchObjectSet.__len__()` <==> *len(x)*

`PitchObjectSet.__lt__()`
x.__lt__(y) <==> x<y

`PitchObjectSet.__ne__()`
x.__ne__(y) <==> x!=y

`PitchObjectSet.__or__()`
x.__or__(y) <==> x|y

`PitchObjectSet.__rand__()`
x.__rand__(y) <==> y&x

`PitchObjectSet.__repr__()` <==> *repr(x)*

`PitchObjectSet.__ror__()`
x.__ror__(y) <==> y|x

`PitchObjectSet.__rsub__()`
x.__rsub__(y) <==> y-x

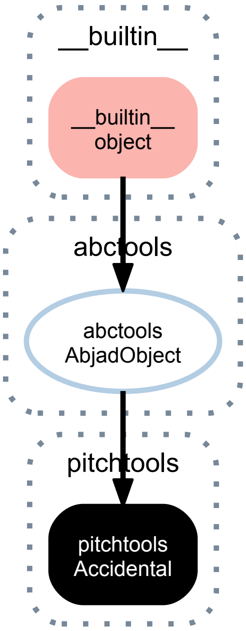
PitchObjectSet.**__rxor__**()
x.__rxor__(y) <==> y^x

PitchObjectSet.**__sub__**()
x.__sub__(y) <==> x-y

PitchObjectSet.**__xor__**()
x.__xor__(y) <==> x^y

21.2 Concrete Classes

21.2.1 pitchtools.Accidental



class pitchtools.**Accidental**(*arg*='')
New in version 2.0. Abjad model of the accidental:

```
>>> pitchtools.Accidental('s')
Accidental('s')
```

Accidentals are immutable.

Read-only properties

Accidental.**alphabetic_accidental_abbreviation**
Read-only alphabetic string:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.alphabetic_accidental_abbreviation
's'
```

Return string.

Accidental.**is_adjusted**
True for all accidentals equal to a nonzero number of semitones. False otherwise:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.is_adjusted
True
```

Return boolean.

Accidental.**lilypond_format**
Read-only LilyPond input format of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.lilypond_format
's'
```

Return string.

Accidental.**name**
Read-only name of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.name
'sharp'
```

Return string.

Accidental.**semitones**
Read-only semitones of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.semitones
1
```

Return number.

Accidental.**storage_format**
Storage format of Abjad object.

Return string.

Accidental.**symbolic_accidental_string**
Read-only symbolic string of accidental:

```
>>> accidental = pitchtools.Accidental('s')
>>> accidental.symbolic_accidental_string
'#'
```

Return string.

Special methods

Accidental.**__add__**(*arg*)

Accidental.**__eq__**(*arg*)

Accidental.**__ge__**(*arg*)

Accidental.**__gt__**(*arg*)

Accidental.**__le__**(*arg*)

Accidental.**__lt__**(*arg*)

Accidental.**__ne__**(*arg*)

Accidental.**__neg__**()

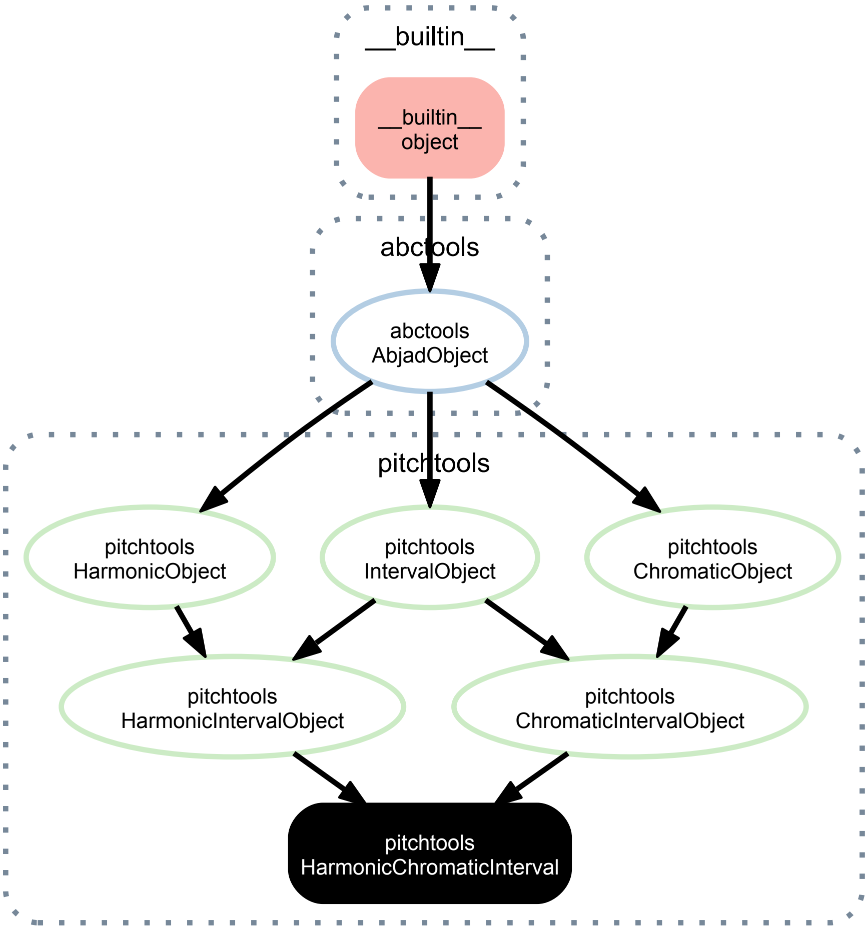
Accidental.**__nonzero__**()

Accidental.**__repr__**()

Accidental.**__str__**()

Accidental.**__sub__**(*arg*)

21.2.2 pitchtools.HarmonicChromaticInterval



class `pitchtools.HarmonicChromaticInterval` (*arg*)

New in version 2.0. Abjad model of harmonic chromatic interval:

```
>>> pitchtools.HarmonicChromaticInterval(-14)
HarmonicChromaticInterval(14)
```

Harmonic chromatic intervals are immutable.

Read-only properties

`HarmonicChromaticInterval.`**`cents`**

`HarmonicChromaticInterval.`**`harmonic_chromatic_interval_class`**

Read-only harmonic chromatic interval-class:

```
>>> harmonic_chromatic_interval = pitchtools.HarmonicChromaticInterval(14)
>>> harmonic_chromatic_interval.harmonic_chromatic_interval_class
HarmonicChromaticIntervalClass(2)
```

Return harmonic chromatic interval-class.

`HarmonicChromaticInterval.`**`interval_class`**

`HarmonicChromaticInterval.`**`number`**

`HarmonicChromaticInterval.`**`semitones`**

`HarmonicChromaticInterval.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

HarmonicChromaticInterval.**__abs__**()

HarmonicChromaticInterval.**__add__**(*arg*)

HarmonicChromaticInterval.**__copy__**()

HarmonicChromaticInterval.**__eq__**(*arg*)

HarmonicChromaticInterval.**__float__**()

HarmonicChromaticInterval.**__ge__**(*arg*)

HarmonicChromaticInterval.**__gt__**(*arg*)

HarmonicChromaticInterval.**__hash__**()

HarmonicChromaticInterval.**__int__**()

HarmonicChromaticInterval.**__le__**(*arg*)

HarmonicChromaticInterval.**__lt__**(*arg*)

HarmonicChromaticInterval.**__ne__**(*arg*)

HarmonicChromaticInterval.**__repr__**()

HarmonicChromaticInterval.**__str__**()

HarmonicChromaticInterval.**__sub__**(*arg*)

21.2.3 pitchtools.HarmonicChromaticIntervalClass

class `pitchtools.`**`HarmonicChromaticIntervalClass`**(*token*)

New in version 2.0. Abjad model of harmonic chromatic interval-class:

```
>>> pitchtools.HarmonicChromaticIntervalClass(-14)
HarmonicChromaticIntervalClass(2)
```

Harmonic chromatic interval-classes are immutable.

Read-only properties

`HarmonicChromaticIntervalClass.`**`number`**

`HarmonicChromaticIntervalClass.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`HarmonicChromaticIntervalClass.`**`__abs__`**()

`HarmonicChromaticIntervalClass.`**`__eq__`**(*arg*)

`HarmonicChromaticIntervalClass.`**`__float__`**()

`HarmonicChromaticIntervalClass.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`HarmonicChromaticIntervalClass.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`HarmonicChromaticIntervalClass.`**`__hash__`**()

`HarmonicChromaticIntervalClass.`**`__int__`**()

`HarmonicChromaticIntervalClass.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`HarmonicChromaticIntervalClass.`**`__lt__`**(*expr*)

Abjad objects by default do not implement this method.

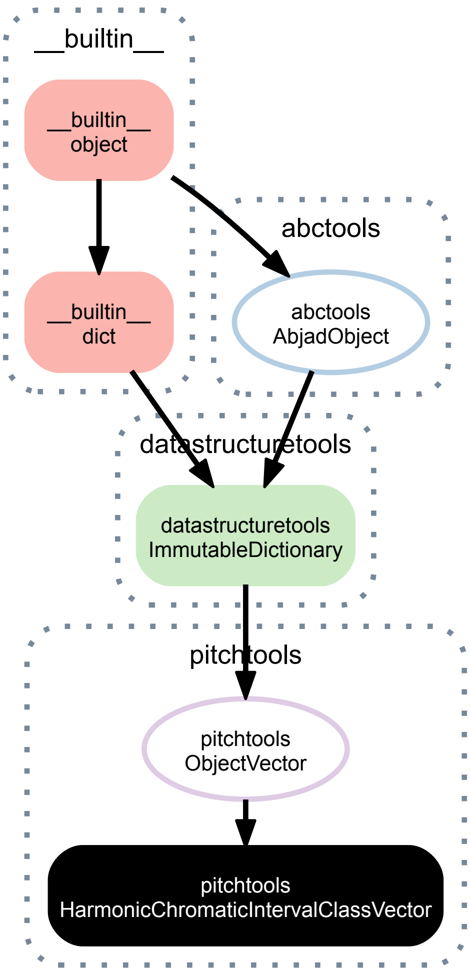
Raise exception.

`HarmonicChromaticIntervalClass.`**`__ne__`**(*arg*)

`HarmonicChromaticIntervalClass.`**`__repr__`**()

`HarmonicChromaticIntervalClass.`**`__str__`**()

21.2.4 pitchtools.HarmonicChromaticIntervalClassVector



class `pitchtools.`**`HarmonicChromaticIntervalClassVector`**(*expr*)

New in version 2.0. Abjad model of harmonic chromatic interval-class vector:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> hcicv = pitchtools.HarmonicChromaticIntervalClassVector(staff)
>>> print hcicv
0 1 3 2 1 2 0 1 0 0 0 0
```

Harmonic chromatic interval-class vector is quartertone-aware:

```
>>> staff.append(Note(1.5, (1, 4)))
>>> hcicv = pitchtools.HarmonicChromaticIntervalClassVector(staff)
>>> print hcicv
0 1 3 2 1 2 0 1 0 0 0 0
1 1 1 1 0 1 0 0 0 0 0 0
```

Harmonic chromatic interval-class vectors are immutable.

Read-only properties

`HarmonicChromaticIntervalClassVector.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`HarmonicChromaticIntervalClassVector.`**`clear`**`()` → None. Remove all items from D.

`HarmonicChromaticIntervalClassVector.`**`copy`**`()` → a shallow copy of D

`HarmonicChromaticIntervalClassVector.`**`get`**`(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

`HarmonicChromaticIntervalClassVector.`**`has_key`**`(`*k*`)` → True if D has a key k, else False

`HarmonicChromaticIntervalClassVector.`**`has_none_of`**`(`*chromatic_interval_numbers*`)`
True when harmonic chromatic interval-class vector contains none of *chromatic_interval_numbers*. Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
```

```
>>> hcicv = pitchtools.HarmonicChromaticIntervalClassVector(staff)
>>> hcicv.has_none_of([9, 10, 11])
True
```

Return boolean.

`HarmonicChromaticIntervalClassVector.`**`items`**`()` → list of D's (key, value) pairs, as 2-tuples

`HarmonicChromaticIntervalClassVector.`**`iteritems`**`()` → an iterator over the (key, value) items of D

`HarmonicChromaticIntervalClassVector.`**`iterkeys`**`()` → an iterator over the keys of D

`HarmonicChromaticIntervalClassVector.`**`itervalues`**`()` → an iterator over the values of D

`HarmonicChromaticIntervalClassVector.`**`keys`**`()` → list of D's keys

`HarmonicChromaticIntervalClassVector.`**`pop`**`(`*k*[, *d*]`)` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`HarmonicChromaticIntervalClassVector.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`HarmonicChromaticIntervalClassVector.`**`setdefault`**`(`*k*[, *d*]`)` → D.get(k,d), also set D[k]=d if k not in D

`HarmonicChromaticIntervalClassVector.`**`update`**`(`[*E*], ***F*`)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`HarmonicChromaticIntervalClassVector.`**`values`**`()` → list of D's values

`HarmonicChromaticIntervalClassVector.`**`viewitems`**`()` → a set-like object providing a view on D's items

`HarmonicChromaticIntervalClassVector.`**`viewkeys`**`()` → a set-like object providing a view on D's keys

`HarmonicChromaticIntervalClassVector.`**`viewvalues`**`()` → an object providing a view on D's values

Special methods

`HarmonicChromaticIntervalClassVector.`**`__cmp__`**`(`*y*`)` <==> *cmp(x, y)*

`HarmonicChromaticIntervalClassVector.`**`__contains__`**`(`*k*`)` → True if D has a key k, else False

`HarmonicChromaticIntervalClassVector.`**`__delitem__`**`(`**args*`)`

HarmonicChromaticIntervalClassVector.**__eq__**()
    x.__eq__(y) <==> x==y

HarmonicChromaticIntervalClassVector.**__ge__**()
    x.__ge__(y) <==> x>=y

HarmonicChromaticIntervalClassVector.**__getitem__**()
    x.__getitem__(y) <==> x[y]

HarmonicChromaticIntervalClassVector.**__gt__**()
    x.__gt__(y) <==> x>y

HarmonicChromaticIntervalClassVector.**__iter__**() <==> *iter(x)*

HarmonicChromaticIntervalClassVector.**__le__**()
    x.__le__(y) <==> x<=y

HarmonicChromaticIntervalClassVector.**__len__**() <==> *len(x)*

HarmonicChromaticIntervalClassVector.**__lt__**()
    x.__lt__(y) <==> x<y

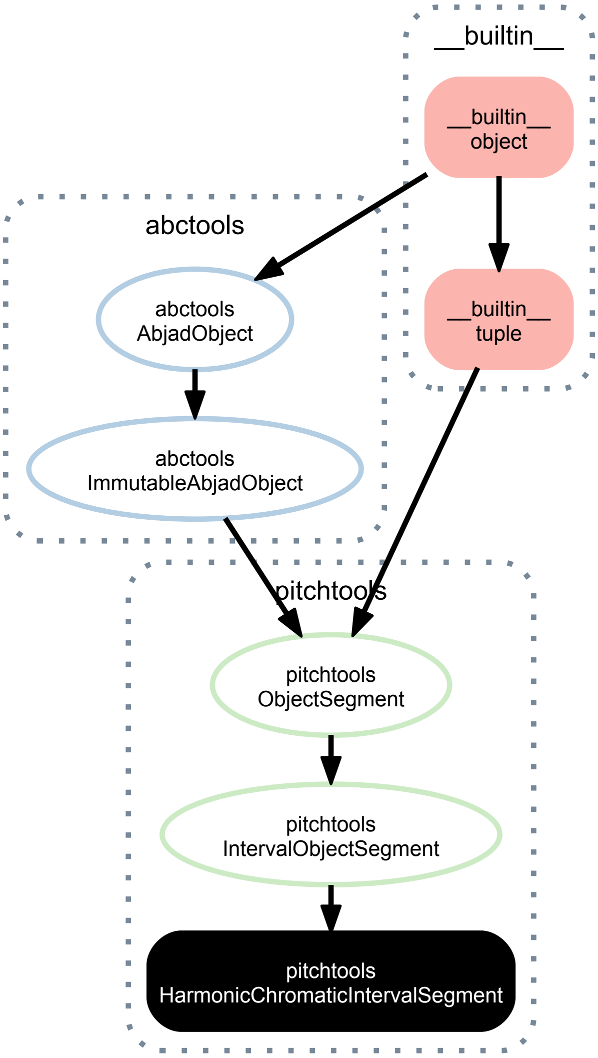
HarmonicChromaticIntervalClassVector.**__ne__**()
    x.__ne__(y) <==> x!=y

HarmonicChromaticIntervalClassVector.**__repr__**()

HarmonicChromaticIntervalClassVector.**__setitem__**(**args*)

HarmonicChromaticIntervalClassVector.**__str__**()

21.2.5 pitchtools.HarmonicChromaticIntervalSegment



class `pitchtools.`**`HarmonicChromaticIntervalSegment`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of harmonic chromatic interval segment:

```
>>> pitchtools.HarmonicChromaticIntervalSegment([10, -12, -13, -13.5])
HarmonicChromaticIntervalSegment(10, 12, 13, 13.5)
```

Harmonic chromatic interval segments are immutable.

Read-only properties

`HarmonicChromaticIntervalSegment.`**`interval_classes`**

`HarmonicChromaticIntervalSegment.`**`intervals`**

`HarmonicChromaticIntervalSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`HarmonicChromaticIntervalSegment.`**`count`**(*value*) → integer – return number of occurrences of value

HarmonicChromaticIntervalSegment.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

HarmonicChromaticIntervalSegment.**rotate**(*n*)

Special methods

HarmonicChromaticIntervalSegment.**__add__**(*arg*)

HarmonicChromaticIntervalSegment.**__contains__**()
x.__contains__(y) <==> y in x

HarmonicChromaticIntervalSegment.**__eq__**()
x.__eq__(y) <==> x==y

HarmonicChromaticIntervalSegment.**__ge__**()
x.__ge__(y) <==> x>=y

HarmonicChromaticIntervalSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

HarmonicChromaticIntervalSegment.**__getslice__**(*start*, *stop*)

HarmonicChromaticIntervalSegment.**__gt__**()
x.__gt__(y) <==> x>y

HarmonicChromaticIntervalSegment.**__hash__**() <==> *hash(x)*

HarmonicChromaticIntervalSegment.**__iter__**() <==> *iter(x)*

HarmonicChromaticIntervalSegment.**__le__**()
x.__le__(y) <==> x<=y

HarmonicChromaticIntervalSegment.**__len__**() <==> *len(x)*

HarmonicChromaticIntervalSegment.**__lt__**()
x.__lt__(y) <==> x<y

HarmonicChromaticIntervalSegment.**__mul__**(*n*)

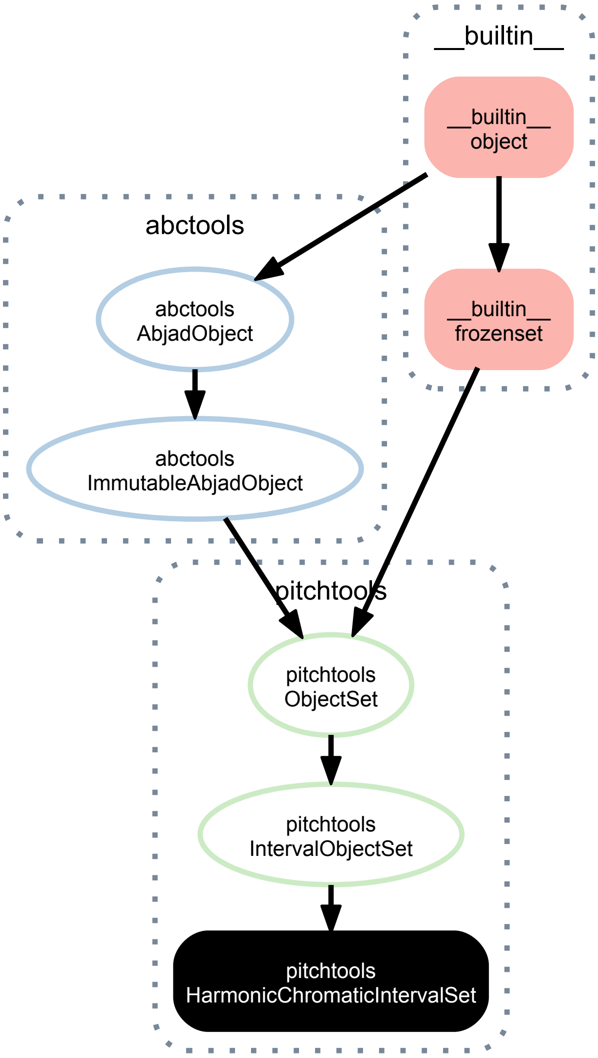
HarmonicChromaticIntervalSegment.**__ne__**()
x.__ne__(y) <==> x!=y

HarmonicChromaticIntervalSegment.**__repr__**()

HarmonicChromaticIntervalSegment.**__rmul__**(*n*)

HarmonicChromaticIntervalSegment.**__str__**()

21.2.6 pitchtools.HarmonicChromaticIntervalSet



class `pitchtools.`**`HarmonicChromaticIntervalSet`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of harmonic chromatic interval set:

```
>>> pitchtools.HarmonicChromaticIntervalSet([10, -12, -13, -13, -13.5])
HarmonicChromaticIntervalSet(10, 12, 13, 13.5)
```

Harmonic chromatic interval sets are immutable.

Read-only properties

`HarmonicChromaticIntervalSet.`**`harmonic_chromatic_interval_numbers`**

`HarmonicChromaticIntervalSet.`**`harmonic_chromatic_intervals`**

`HarmonicChromaticIntervalSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`HarmonicChromaticIntervalSet.`**`copy`**()

Return a shallow copy of a set.

HarmonicChromaticIntervalSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

HarmonicChromaticIntervalSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

HarmonicChromaticIntervalSet.**isdisjoint**()
Return True if two sets have a null intersection.

HarmonicChromaticIntervalSet.**issubset**()
Report whether another set contains this set.

HarmonicChromaticIntervalSet.**issuperset**()
Report whether this set contains another set.

HarmonicChromaticIntervalSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

HarmonicChromaticIntervalSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

HarmonicChromaticIntervalSet.**__and__**()
x.__and__(y) <==> x&y

HarmonicChromaticIntervalSet.**__cmp__**(*y*) <==> *cmp(x, y)*

HarmonicChromaticIntervalSet.**__contains__**()
x.__contains__(y) <==> y in x.

HarmonicChromaticIntervalSet.**__copy__**()

HarmonicChromaticIntervalSet.**__eq__**()
x.__eq__(y) <==> x==y

HarmonicChromaticIntervalSet.**__ge__**()
x.__ge__(y) <==> x>=y

HarmonicChromaticIntervalSet.**__gt__**()
x.__gt__(y) <==> x>y

HarmonicChromaticIntervalSet.**__hash__**() <==> *hash(x)*

HarmonicChromaticIntervalSet.**__iter__**() <==> *iter(x)*

HarmonicChromaticIntervalSet.**__le__**()
x.__le__(y) <==> x<=y

HarmonicChromaticIntervalSet.**__len__**() <==> *len(x)*

HarmonicChromaticIntervalSet.**__lt__**()
x.__lt__(y) <==> x<y

HarmonicChromaticIntervalSet.**__ne__**()
x.__ne__(y) <==> x!=y

HarmonicChromaticIntervalSet.**__or__**()
x.__or__(y) <==> x|y

HarmonicChromaticIntervalSet.**__rand__**()
x.__rand__(y) <==> y&x

HarmonicChromaticIntervalSet.**__repr__**()

HarmonicChromaticIntervalSet.**__ror__**()
x.__ror__(y) <==> y|x

HarmonicChromaticIntervalSet.**__rsub__**()
x.__rsub__(y) <==> y-x

HarmonicChromaticIntervalSet.**__rxor__**()
x.__rxor__(y) <==> y^x

HarmonicChromaticIntervalSet.**__str__**()

HarmonicChromaticIntervalSet.**__sub__**()
x.__sub__(y) <==> x-y

HarmonicChromaticIntervalSet.**__xor__**()
x.__xor__(y) <==> x^y

21.2.7 pitchtools.HarmonicCounterpointInterval

class pitchtools.**HarmonicCounterpointInterval**(*token*)
New in version 2.0. Abjad model of harmonic counterpoint interval:

```
>>> pitchtools.HarmonicCounterpointInterval(-9)
HarmonicCounterpointInterval(9)
```

Harmonic counterpoint intervals are immutable.

Read-only properties

HarmonicCounterpointInterval.**cents**

HarmonicCounterpointInterval.**harmonic_counterpoint_interval_class**

HarmonicCounterpointInterval.**interval_class**

HarmonicCounterpointInterval.**number**

HarmonicCounterpointInterval.**semitones**

HarmonicCounterpointInterval.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

HarmonicCounterpointInterval.**__abs__**()

HarmonicCounterpointInterval.**__eq__**(*arg*)

HarmonicCounterpointInterval.**__float__**()

HarmonicCounterpointInterval.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicCounterpointInterval.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HarmonicCounterpointInterval.**__hash__**()

HarmonicCounterpointInterval.**__int__**()

HarmonicCounterpointInterval.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicCounterpointInterval.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicCounterpointInterval.**__ne__**(*arg*)

HarmonicCounterpointInterval.**__repr__**()

HarmonicCounterpointInterval.**__str__**()

21.2.8 pitchtools.HarmonicCounterpointIntervalClass

class `pitchtools.`**`HarmonicCounterpointIntervalClass`**(*token*)

New in version 2.0. Abjad model of harmonic counterpoint interval-class:

```
>>> pitchtools.HarmonicCounterpointIntervalClass(-9)
HarmonicCounterpointIntervalClass(2)
```

Harmonic counterpoint interval-classes are immutable.

Read-only properties

`HarmonicCounterpointIntervalClass.`**`number`**

`HarmonicCounterpointIntervalClass.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`HarmonicCounterpointIntervalClass.`**`__abs__`**()

`HarmonicCounterpointIntervalClass.`**`__eq__`**(*arg*)

`HarmonicCounterpointIntervalClass.`**`__float__`**()

`HarmonicCounterpointIntervalClass.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`HarmonicCounterpointIntervalClass.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`HarmonicCounterpointIntervalClass.__hash__()`

`HarmonicCounterpointIntervalClass.__int__()`

`HarmonicCounterpointIntervalClass.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`HarmonicCounterpointIntervalClass.__lt__(expr)`
Abjad objects by default do not implement this method.

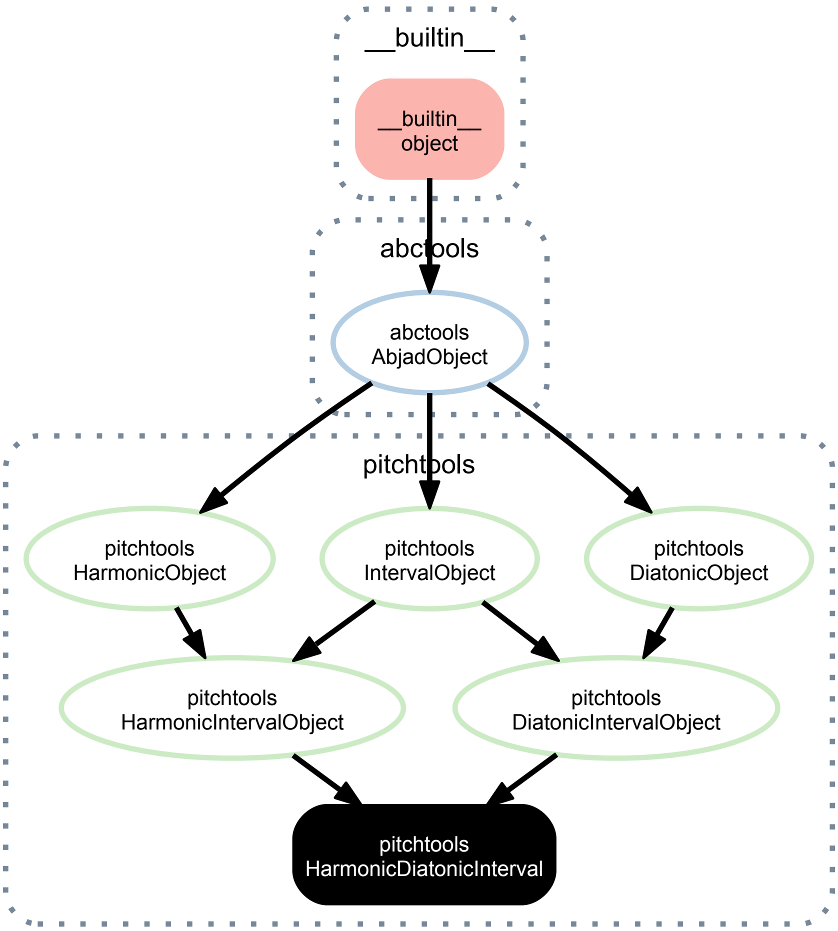
Raise exception.

`HarmonicCounterpointIntervalClass.__ne__(arg)`

`HarmonicCounterpointIntervalClass.__repr__()`

`HarmonicCounterpointIntervalClass.__str__()`

21.2.9 pitchtools.HarmonicDiatonicInterval



class `pitchtools.HarmonicDiatonicInterval(*args)`
New in version 2.0. Abjad model harmonic diatonic interval:

```
>>> pitchtools.HarmonicDiatonicInterval('M9')
HarmonicDiatonicInterval('M9')
```

Harmonic diatonic intervals are immutable.

Read-only properties

HarmonicDiatonicInterval.**cents**

HarmonicDiatonicInterval.**diatonic_interval_class**

HarmonicDiatonicInterval.**harmonic_counterpoint_interval**

HarmonicDiatonicInterval.**harmonic_diatonic_interval_class**

HarmonicDiatonicInterval.**interval_class**

HarmonicDiatonicInterval.**interval_string**

HarmonicDiatonicInterval.**melodic_diatonic_interval_ascending**

HarmonicDiatonicInterval.**melodic_diatonic_interval_descending**

HarmonicDiatonicInterval.**number**

HarmonicDiatonicInterval.**quality_string**

HarmonicDiatonicInterval.**semitones**

HarmonicDiatonicInterval.**staff_spaces**

HarmonicDiatonicInterval.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

HarmonicDiatonicInterval.**__abs__**()

HarmonicDiatonicInterval.**__copy__**(**args*)

HarmonicDiatonicInterval.**__deepcopy__**(**args*)

HarmonicDiatonicInterval.**__eq__**(*arg*)

HarmonicDiatonicInterval.**__float__**()

HarmonicDiatonicInterval.**__ge__**(*arg*)

HarmonicDiatonicInterval.**__gt__**(*arg*)

HarmonicDiatonicInterval.**__hash__**()

HarmonicDiatonicInterval.**__int__**()

HarmonicDiatonicInterval.**__le__**(*arg*)

HarmonicDiatonicInterval.**__lt__**(*arg*)

HarmonicDiatonicInterval.**__ne__**(*arg*)

HarmonicDiatonicInterval.**__repr__**()

HarmonicDiatonicInterval.**__str__**()

21.2.10 pitchtools.HarmonicDiatonicIntervalClass

class `pitchtools.`**`HarmonicDiatonicIntervalClass`**(**args*)

New in version 2.0. Abjad model harmonic diatonic interval-class:

```
>>> pitchtools.HarmonicDiatonicIntervalClass('-M9')
HarmonicDiatonicIntervalClass('M2')
```

Harmonic diatonic interval-classes are immutable.

Read-only properties

`HarmonicDiatonicIntervalClass.`**`number`**

`HarmonicDiatonicIntervalClass.`**`quality_string`**

`HarmonicDiatonicIntervalClass.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`HarmonicDiatonicIntervalClass.`**`invert`**()

Read-only inversion of harmonic diatonic interval-class:

```
>>> hdic = pitchtools.HarmonicDiatonicIntervalClass('major', -9)
>>> hdic.invert()
HarmonicDiatonicIntervalClass('m7')
```

Return harmonic diatonic interval-class.

Special methods

HarmonicDiatonicIntervalClass.**__abs__**()

HarmonicDiatonicIntervalClass.**__eq__**(*arg*)

HarmonicDiatonicIntervalClass.**__float__**()

HarmonicDiatonicIntervalClass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicDiatonicIntervalClass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HarmonicDiatonicIntervalClass.**__hash__**()

HarmonicDiatonicIntervalClass.**__int__**()

HarmonicDiatonicIntervalClass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HarmonicDiatonicIntervalClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

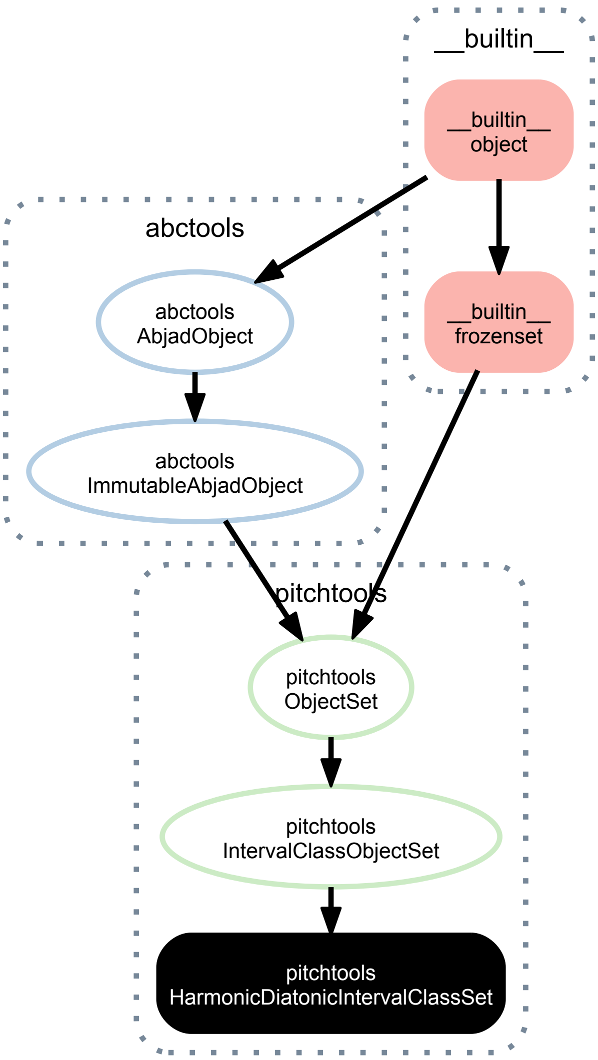
Raise exception.

HarmonicDiatonicIntervalClass.**__ne__**(*arg*)

HarmonicDiatonicIntervalClass.**__repr__**()

HarmonicDiatonicIntervalClass.**__str__**()

21.2.11 pitchtools.HarmonicDiatonicIntervalClassSet



class `pitchtools.`**`HarmonicDiatonicIntervalClassSet`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of harmonic diatonic interval-class set:

```
>>> pitchtools.HarmonicDiatonicIntervalClassSet('m2 M2 m3 M3')
HarmonicDiatonicIntervalClassSet('m2 M2 m3 M3')
```

Harmonic diatonic interval-class sets are immutable.

Read-only properties

`HarmonicDiatonicIntervalClassSet.`**`harmonic_diatonic_interval_classes`**

`HarmonicDiatonicIntervalClassSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`HarmonicDiatonicIntervalClassSet.`**`copy`**()

Return a shallow copy of a set.

HarmonicDiatonicIntervalClassSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

HarmonicDiatonicIntervalClassSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

HarmonicDiatonicIntervalClassSet.**isdisjoint**()
Return True if two sets have a null intersection.

HarmonicDiatonicIntervalClassSet.**issubset**()
Report whether another set contains this set.

HarmonicDiatonicIntervalClassSet.**issuperset**()
Report whether this set contains another set.

HarmonicDiatonicIntervalClassSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

HarmonicDiatonicIntervalClassSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

HarmonicDiatonicIntervalClassSet.**__and__**()
x.__and__(y) <==> x&y

HarmonicDiatonicIntervalClassSet.**__cmp__**(*y*) <==> *cmp(x, y)*

HarmonicDiatonicIntervalClassSet.**__contains__**()
x.__contains__(y) <==> y in x.

HarmonicDiatonicIntervalClassSet.**__copy__**()

HarmonicDiatonicIntervalClassSet.**__eq__**()
x.__eq__(y) <==> x==y

HarmonicDiatonicIntervalClassSet.**__ge__**()
x.__ge__(y) <==> x>=y

HarmonicDiatonicIntervalClassSet.**__gt__**()
x.__gt__(y) <==> x>y

HarmonicDiatonicIntervalClassSet.**__hash__**() <==> *hash(x)*

HarmonicDiatonicIntervalClassSet.**__iter__**() <==> *iter(x)*

HarmonicDiatonicIntervalClassSet.**__le__**()
x.__le__(y) <==> x<=y

HarmonicDiatonicIntervalClassSet.**__len__**() <==> *len(x)*

HarmonicDiatonicIntervalClassSet.**__lt__**()
x.__lt__(y) <==> x<y

HarmonicDiatonicIntervalClassSet.**__ne__**()
x.__ne__(y) <==> x!=y

HarmonicDiatonicIntervalClassSet.**__or__**()
x.__or__(y) <==> x|y

`HarmonicDiatonicIntervalClassSet.__rand__()`
x.__rand__(y) <==> y&x

`HarmonicDiatonicIntervalClassSet.__repr__()`

`HarmonicDiatonicIntervalClassSet.__ror__()`
x.__ror__(y) <==> y|x

`HarmonicDiatonicIntervalClassSet.__rsub__()`
x.__rsub__(y) <==> y-x

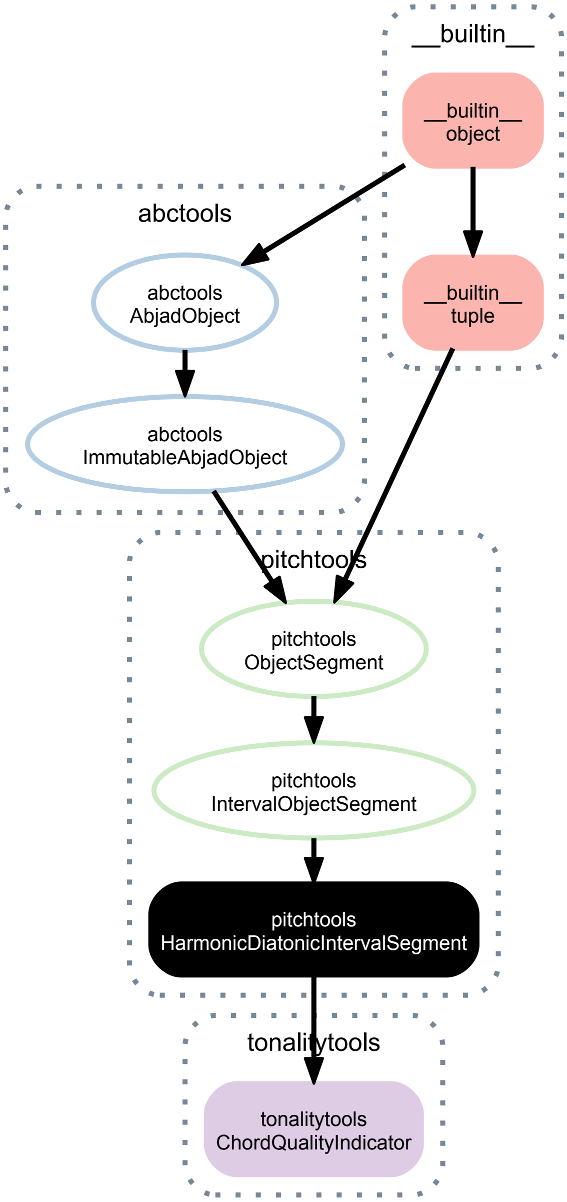
`HarmonicDiatonicIntervalClassSet.__rxor__()`
x.__rxor__(y) <==> y^x

`HarmonicDiatonicIntervalClassSet.__str__()`

`HarmonicDiatonicIntervalClassSet.__sub__()`
x.__sub__(y) <==> x-y

`HarmonicDiatonicIntervalClassSet.__xor__()`
x.__xor__(y) <==> x^y

21.2.12 pitchtools.HarmonicDiatonicIntervalSegment



class `pitchtools.`**`HarmonicDiatonicIntervalSegment`** (**args*, ***kwargs*)

New in version 2.0. Abjad model of harmonic diatonic interval segment:

```
>>> pitchtools.HarmonicDiatonicIntervalSegment('m2 M9 m3 M3')
HarmonicDiatonicIntervalSegment('m2 M9 m3 M3')
```

Harmonic diatonic interval segments are immutable.

Read-only properties

`HarmonicDiatonicIntervalSegment.`**`harmonic_chromatic_interval_segment`**

`HarmonicDiatonicIntervalSegment.`**`interval_classes`**

`HarmonicDiatonicIntervalSegment.`**`intervals`**

`HarmonicDiatonicIntervalSegment.`**`melodic_chromatic_interval_segment`**

`HarmonicDiatonicIntervalSegment.`**`melodic_diatonic_interval_segment`**

HarmonicDiatonicIntervalSegment.**storage_format**
Storage format of Abjad object.

Return string.

Methods

HarmonicDiatonicIntervalSegment.**count**(*value*) → integer – return number of occurrences of value

HarmonicDiatonicIntervalSegment.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

HarmonicDiatonicIntervalSegment.**rotate**(*n*)

Special methods

HarmonicDiatonicIntervalSegment.**__add__**(*arg*)

HarmonicDiatonicIntervalSegment.**__contains__**()
x.__contains__(y) <==> y in x

HarmonicDiatonicIntervalSegment.**__copy__**()

HarmonicDiatonicIntervalSegment.**__eq__**()
x.__eq__(y) <==> x==y

HarmonicDiatonicIntervalSegment.**__ge__**()
x.__ge__(y) <==> x>=y

HarmonicDiatonicIntervalSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

HarmonicDiatonicIntervalSegment.**__getslice__**(*start*, *stop*)

HarmonicDiatonicIntervalSegment.**__gt__**()
x.__gt__(y) <==> x>y

HarmonicDiatonicIntervalSegment.**__hash__**() <==> *hash(x)*

HarmonicDiatonicIntervalSegment.**__iter__**() <==> *iter(x)*

HarmonicDiatonicIntervalSegment.**__le__**()
x.__le__(y) <==> x<=y

HarmonicDiatonicIntervalSegment.**__len__**() <==> *len(x)*

HarmonicDiatonicIntervalSegment.**__lt__**()
x.__lt__(y) <==> x<y

HarmonicDiatonicIntervalSegment.**__mul__**(*n*)

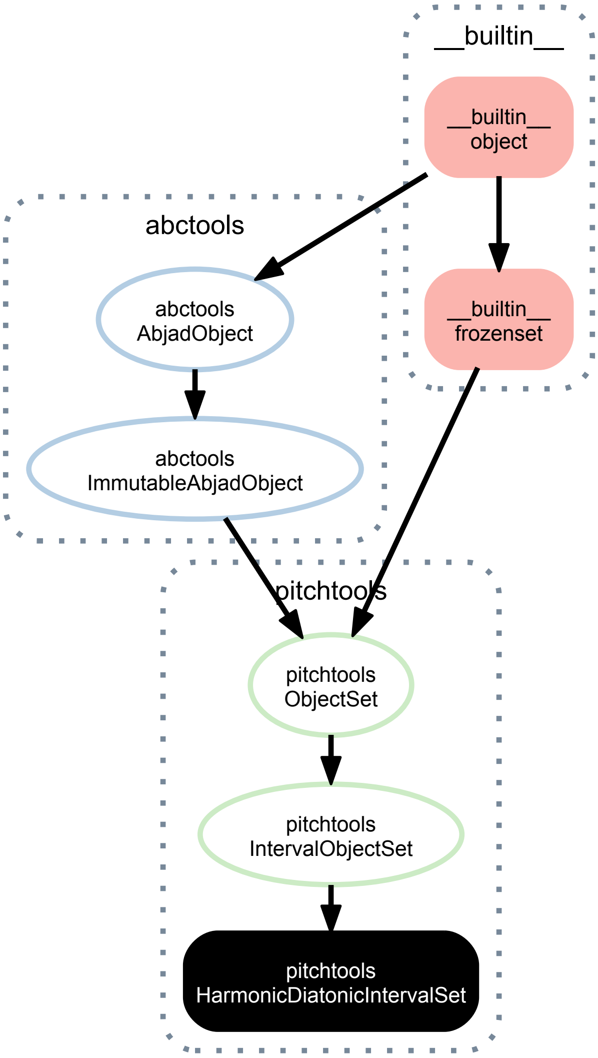
HarmonicDiatonicIntervalSegment.**__ne__**()
x.__ne__(y) <==> x!=y

HarmonicDiatonicIntervalSegment.**__repr__**()

HarmonicDiatonicIntervalSegment.**__rmul__**(*n*)

HarmonicDiatonicIntervalSegment.**__str__**()

21.2.13 pitchtools.HarmonicDiatonicIntervalSet



class `pitchtools.HarmonicDiatonicIntervalSet` (**args*, ***kwargs*)

New in version 2.0. Abjad model of harmonic diatonic interval set:

```
>>> pitchtools.HarmonicDiatonicIntervalSet('m2 m2 M2 M9')
HarmonicDiatonicIntervalSet('m2 M2 M9')
```

Harmonic diatonic interval sets are immutable.

Read-only properties

`HarmonicDiatonicIntervalSet.`**`harmonic_chromatic_interval_set`**

`HarmonicDiatonicIntervalSet.`**`harmonic_diatonic_interval_numbers`**

`HarmonicDiatonicIntervalSet.`**`harmonic_diatonic_intervals`**

`HarmonicDiatonicIntervalSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

HarmonicDiatonicIntervalSet.**copy**()
Return a shallow copy of a set.

HarmonicDiatonicIntervalSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

HarmonicDiatonicIntervalSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

HarmonicDiatonicIntervalSet.**isdisjoint**()
Return True if two sets have a null intersection.

HarmonicDiatonicIntervalSet.**issubset**()
Report whether another set contains this set.

HarmonicDiatonicIntervalSet.**issuperset**()
Report whether this set contains another set.

HarmonicDiatonicIntervalSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

HarmonicDiatonicIntervalSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

HarmonicDiatonicIntervalSet.**__and__**()
x.__and__(y) <==> x&y

HarmonicDiatonicIntervalSet.**__cmp__**(*y) <==> cmp(x, y*)

HarmonicDiatonicIntervalSet.**__contains__**()
x.__contains__(y) <==> y in x.

HarmonicDiatonicIntervalSet.**__copy__**()

HarmonicDiatonicIntervalSet.**__eq__**()
x.__eq__(y) <==> x==y

HarmonicDiatonicIntervalSet.**__ge__**()
x.__ge__(y) <==> x>=y

HarmonicDiatonicIntervalSet.**__gt__**()
x.__gt__(y) <==> x>y

HarmonicDiatonicIntervalSet.**__hash__**(*) <==> hash(x*)

HarmonicDiatonicIntervalSet.**__iter__**(*) <==> iter(x*)

HarmonicDiatonicIntervalSet.**__le__**()
x.__le__(y) <==> x<=y

HarmonicDiatonicIntervalSet.**__len__**(*) <==> len(x*)

HarmonicDiatonicIntervalSet.**__lt__**()
x.__lt__(y) <==> x<y

HarmonicDiatonicIntervalSet.**__ne__**()
x.__ne__(y) <==> x!=y

HarmonicDiatonicIntervalSet.**__or__**()
    x.__or__(y) <==> x|y

HarmonicDiatonicIntervalSet.**__rand__**()
    x.__rand__(y) <==> y&x

HarmonicDiatonicIntervalSet.**__repr__**()

HarmonicDiatonicIntervalSet.**__ror__**()
    x.__ror__(y) <==> y|x

HarmonicDiatonicIntervalSet.**__rsub__**()
    x.__rsub__(y) <==> y-x

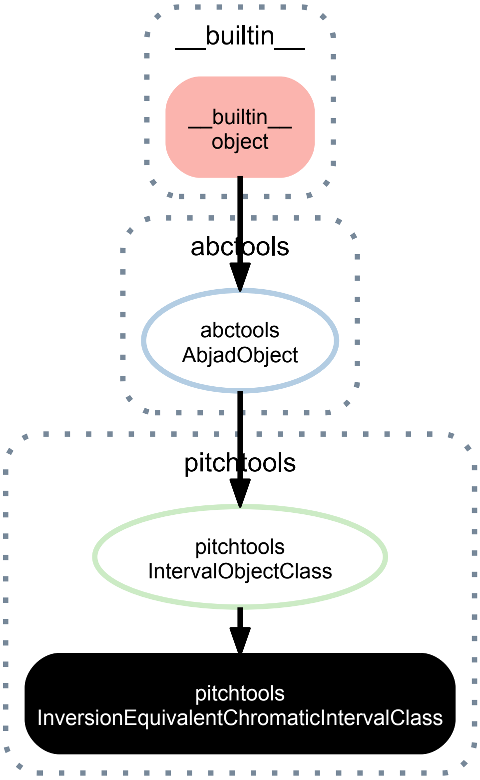
HarmonicDiatonicIntervalSet.**__rxor__**()
    x.__rxor__(y) <==> y^x

HarmonicDiatonicIntervalSet.**__str__**()

HarmonicDiatonicIntervalSet.**__sub__**()
    x.__sub__(y) <==> x-y

HarmonicDiatonicIntervalSet.**__xor__**()
    x.__xor__(y) <==> x^y

21.2.14 pitchtools.InversionEquivalentChromaticIntervalClass



class pitchtools.**InversionEquivalentChromaticIntervalClass**(*interval_class_token*)
    New in version 2.0. Abjad model of inversion-equivalent chromatic interval-class:

```
>>> pitchtools.InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(1)
```

Inversion-equivalent chromatic interval-classes are immutable.

Read-only properties

InversionEquivalentChromaticIntervalClass.**inversion_equivalent_chromatic_interval_number**

InversionEquivalentChromaticIntervalClass.**number**

InversionEquivalentChromaticIntervalClass.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

InversionEquivalentChromaticIntervalClass.**__abs__**()

InversionEquivalentChromaticIntervalClass.**__copy__**()

InversionEquivalentChromaticIntervalClass.**__eq__**(*arg*)

InversionEquivalentChromaticIntervalClass.**__float__**()

InversionEquivalentChromaticIntervalClass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InversionEquivalentChromaticIntervalClass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

InversionEquivalentChromaticIntervalClass.**__hash__**()

InversionEquivalentChromaticIntervalClass.**__int__**()

InversionEquivalentChromaticIntervalClass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InversionEquivalentChromaticIntervalClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

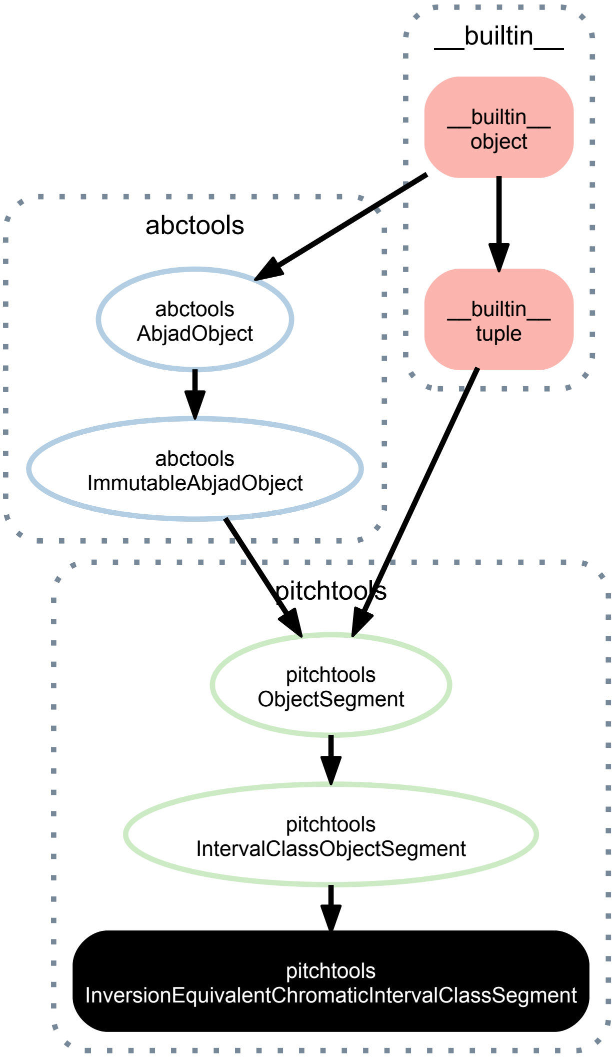
InversionEquivalentChromaticIntervalClass.**__ne__**(*arg*)

InversionEquivalentChromaticIntervalClass.**__neg__**()

InversionEquivalentChromaticIntervalClass.**__repr__**()

InversionEquivalentChromaticIntervalClass.**__str__**()

21.2.15 pitchtools.InversionEquivalentChromaticIntervalClassSegment



class pitchtools.**InversionEquivalentChromaticIntervalClassSegment**(**args*, ***kwargs*)

New in version 2.0. Abjad model of inversion-equivalent chromatic interval-class segment:

```
>>> pitchtools.InversionEquivalentChromaticIntervalClassSegment([2, 1, 0, 5.5, 6])
InversionEquivalentChromaticIntervalClassSegment(2, 1, 0, 5.5, 6)
```

Inversion-equivalent chromatic interval-class segments are immutable.

Read-only properties

InversionEquivalentChromaticIntervalClassSegment.**interval_class_numbers**

InversionEquivalentChromaticIntervalClassSegment.**interval_classes**

InversionEquivalentChromaticIntervalClassSegment.**storage_format**

Storage format of Abjad object.

Return string.

Methods

`InversionEquivalentChromaticIntervalClassSegment.`**count**(*value*) → integer – return number of occurrences of value

`InversionEquivalentChromaticIntervalClassSegment.`**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Special methods

`InversionEquivalentChromaticIntervalClassSegment.`**__add__**(*arg*)

`InversionEquivalentChromaticIntervalClassSegment.`**__contains__**()
x.__contains__(y) <==> y in x

`InversionEquivalentChromaticIntervalClassSegment.`**__eq__**()
x.__eq__(y) <==> x==y

`InversionEquivalentChromaticIntervalClassSegment.`**__ge__**()
x.__ge__(y) <==> x>=y

`InversionEquivalentChromaticIntervalClassSegment.`**__getitem__**()
x.__getitem__(y) <==> x[y]

`InversionEquivalentChromaticIntervalClassSegment.`**__getslice__**(*start*, *stop*)

`InversionEquivalentChromaticIntervalClassSegment.`**__gt__**()
x.__gt__(y) <==> x>y

`InversionEquivalentChromaticIntervalClassSegment.`**__hash__**() <==> *hash(x)*

`InversionEquivalentChromaticIntervalClassSegment.`**__iter__**() <==> *iter(x)*

`InversionEquivalentChromaticIntervalClassSegment.`**__le__**()
x.__le__(y) <==> x<=y

`InversionEquivalentChromaticIntervalClassSegment.`**__len__**() <==> *len(x)*

`InversionEquivalentChromaticIntervalClassSegment.`**__lt__**()
x.__lt__(y) <==> x<y

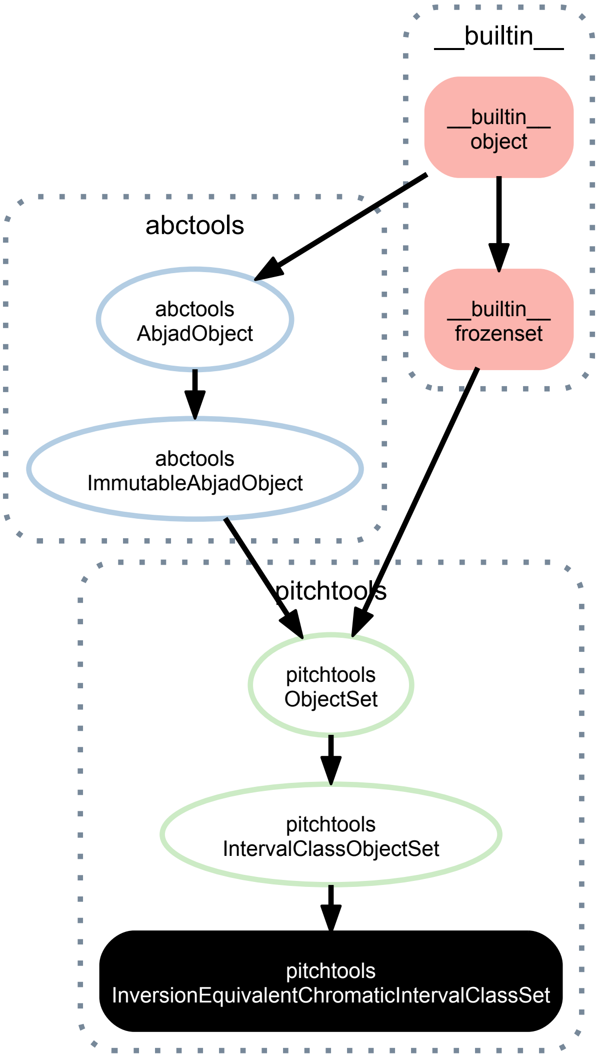
`InversionEquivalentChromaticIntervalClassSegment.`**__mul__**(*n*)

`InversionEquivalentChromaticIntervalClassSegment.`**__ne__**()
x.__ne__(y) <==> x!=y

`InversionEquivalentChromaticIntervalClassSegment.`**__repr__**()

`InversionEquivalentChromaticIntervalClassSegment.`**__rmul__**(*n*)

21.2.16 pitchtools.InversionEquivalentChromaticIntervalClassSet



class `pitchtools.`**`InversionEquivalentChromaticIntervalClassSet`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of inversion-equivalent chromatic interval-class set:

```
>>> pitchtools.InversionEquivalentChromaticIntervalClassSet([1, 1, 6, 2, 2])
InversionEquivalentChromaticIntervalClassSet(1, 2, 6)
```

Inversion-equivalent chromatic interval-class sets are immutable.

Read-only properties

`InversionEquivalentChromaticIntervalClassSet.`**`inversion_equivalent_chromatic_interval_cla`**

`InversionEquivalentChromaticIntervalClassSet.`**`inversion_equivalent_chromatic_interval_cla`**

`InversionEquivalentChromaticIntervalClassSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`InversionEquivalentChromaticIntervalClassSet.`**copy**()
Return a shallow copy of a set.

`InversionEquivalentChromaticIntervalClassSet.`**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`InversionEquivalentChromaticIntervalClassSet.`**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`InversionEquivalentChromaticIntervalClassSet.`**isdisjoint**()
Return True if two sets have a null intersection.

`InversionEquivalentChromaticIntervalClassSet.`**issubset**()
Report whether another set contains this set.

`InversionEquivalentChromaticIntervalClassSet.`**issuperset**()
Report whether this set contains another set.

`InversionEquivalentChromaticIntervalClassSet.`**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`InversionEquivalentChromaticIntervalClassSet.`**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`InversionEquivalentChromaticIntervalClassSet.`**__and__**()
x.__and__(y) <==> x&y

`InversionEquivalentChromaticIntervalClassSet.`**__cmp__**(*y*) <==> *cmp(x, y)*

`InversionEquivalentChromaticIntervalClassSet.`**__contains__**()
x.__contains__(y) <==> y in x.

`InversionEquivalentChromaticIntervalClassSet.`**__copy__**()

`InversionEquivalentChromaticIntervalClassSet.`**__eq__**()
x.__eq__(y) <==> x==y

`InversionEquivalentChromaticIntervalClassSet.`**__ge__**()
x.__ge__(y) <==> x>=y

`InversionEquivalentChromaticIntervalClassSet.`**__gt__**()
x.__gt__(y) <==> x>y

`InversionEquivalentChromaticIntervalClassSet.`**__hash__**() <==> *hash(x)*

`InversionEquivalentChromaticIntervalClassSet.`**__iter__**() <==> *iter(x)*

`InversionEquivalentChromaticIntervalClassSet.`**__le__**()
x.__le__(y) <==> x<=y

`InversionEquivalentChromaticIntervalClassSet.`**__len__**() <==> *len(x)*

`InversionEquivalentChromaticIntervalClassSet.`**__lt__**()
x.__lt__(y) <==> x<y

`InversionEquivalentChromaticIntervalClassSet.`**__ne__**()
x.__ne__(y) <==> x!=y

InversionEquivalentChromaticIntervalClassSet.**__or__**()
x.__or__(y) <==> x|y

InversionEquivalentChromaticIntervalClassSet.**__rand__**()
x.__rand__(y) <==> y&x

InversionEquivalentChromaticIntervalClassSet.**__repr__**()

InversionEquivalentChromaticIntervalClassSet.**__ror__**()
x.__ror__(y) <==> y|x

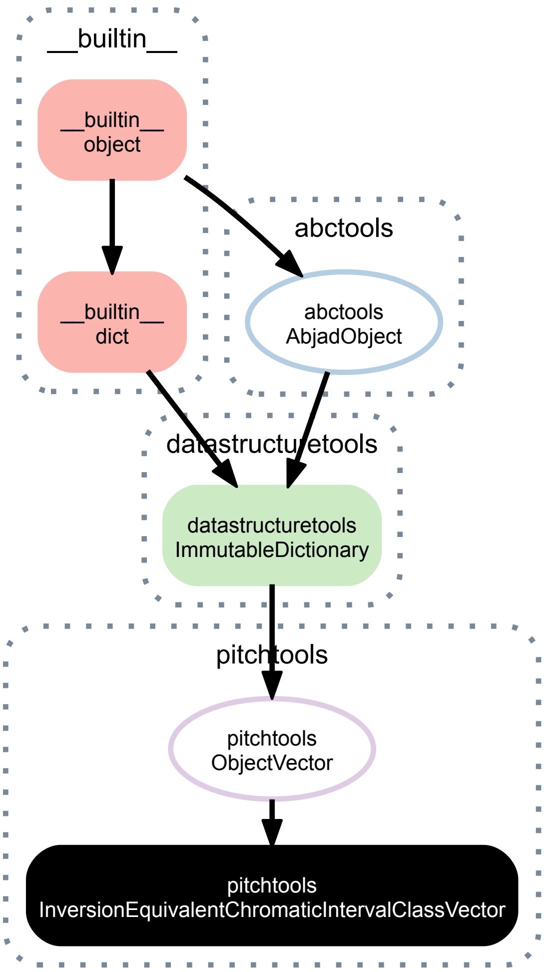
InversionEquivalentChromaticIntervalClassSet.**__rsub__**()
x.__rsub__(y) <==> y-x

InversionEquivalentChromaticIntervalClassSet.**__rxor__**()
x.__rxor__(y) <==> y^x

InversionEquivalentChromaticIntervalClassSet.**__sub__**()
x.__sub__(y) <==> x-y

InversionEquivalentChromaticIntervalClassSet.**__xor__**()
x.__xor__(y) <==> x^y

21.2.17 pitchtools.InversionEquivalentChromaticIntervalClassVector



class pitchtools.**InversionEquivalentChromaticIntervalClassVector**(**args*, ***kwargs*)
New in version 2.0. Abjad model of inversion-equivalent chromatic interval-class vector:

```
>>> pitchtools.InversionEquivalentChromaticIntervalClassVector([1, 1, 6, 2, 2, 2])
InversionEquivalentChromaticIntervalClassVector(0 | 2 3 0 0 0 1)
```

Initialize by inversion-equivalent chromatic interval-class counts:

```
>>> pitchtools.InversionEquivalentChromaticIntervalClassVector(counts=[2, 3, 0, 0, 0, 1])
InversionEquivalentChromaticIntervalClassVector(0 | 2 3 0 0 0 1)
```

Inversion-equivalent chromatic interval-class vectors are immutable.

Read-only properties

`InversionEquivalentChromaticIntervalClassVector.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`InversionEquivalentChromaticIntervalClassVector.`**`clear`**`()` → None. Remove all items from D.

`InversionEquivalentChromaticIntervalClassVector.`**`copy`**`()` → a shallow copy of D

`InversionEquivalentChromaticIntervalClassVector.`**`get`**`(k[, d])` → D[k] if k in D, else d. d defaults to None.

`InversionEquivalentChromaticIntervalClassVector.`**`has_key`**`(k)` → True if D has a key k, else False

`InversionEquivalentChromaticIntervalClassVector.`**`items`**`()` → list of D's (key, value) pairs, as 2-tuples

`InversionEquivalentChromaticIntervalClassVector.`**`iteritems`**`()` → an iterator over the (key, value) items of D

`InversionEquivalentChromaticIntervalClassVector.`**`iterkeys`**`()` → an iterator over the keys of D

`InversionEquivalentChromaticIntervalClassVector.`**`itervalues`**`()` → an iterator over the values of D

`InversionEquivalentChromaticIntervalClassVector.`**`keys`**`()` → list of D's keys

`InversionEquivalentChromaticIntervalClassVector.`**`pop`**`(k[, d])` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`InversionEquivalentChromaticIntervalClassVector.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`InversionEquivalentChromaticIntervalClassVector.`**`setdefault`**`(k[, d])` → D.get(k,d), also set D[k]=d if k not in D

`InversionEquivalentChromaticIntervalClassVector.`**`update`**`([E], **F)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

InversionEquivalentChromaticIntervalClassVector.**values**() → list of D's values

InversionEquivalentChromaticIntervalClassVector.**viewitems**() → a set-like object providing a view on D's items

InversionEquivalentChromaticIntervalClassVector.**viewkeys**() → a set-like object providing a view on D's keys

InversionEquivalentChromaticIntervalClassVector.**viewvalues**() → an object providing a view on D's values

Special methods

InversionEquivalentChromaticIntervalClassVector.**__cmp__**(*y*) <==> *cmp(x, y)*

InversionEquivalentChromaticIntervalClassVector.**__contains__**(*k*) → True if D has a key k, else False

InversionEquivalentChromaticIntervalClassVector.**__delitem__**(**args*)

InversionEquivalentChromaticIntervalClassVector.**__eq__**()
x.__eq__(y) <==> x==y

InversionEquivalentChromaticIntervalClassVector.**__ge__**()
x.__ge__(y) <==> x>=y

InversionEquivalentChromaticIntervalClassVector.**__getitem__**()
x.__getitem__(y) <==> x[y]

InversionEquivalentChromaticIntervalClassVector.**__gt__**()
x.__gt__(y) <==> x>y

InversionEquivalentChromaticIntervalClassVector.**__iter__**() <==> *iter(x)*

InversionEquivalentChromaticIntervalClassVector.**__le__**()
x.__le__(y) <==> x<=y

InversionEquivalentChromaticIntervalClassVector.**__len__**() <==> *len(x)*

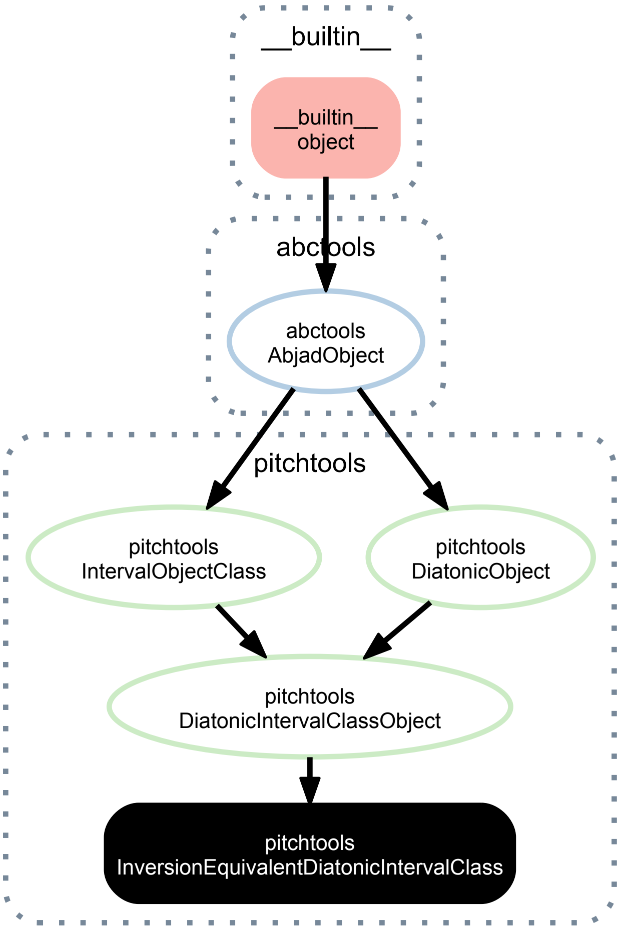
InversionEquivalentChromaticIntervalClassVector.**__lt__**()
x.__lt__(y) <==> x<y

InversionEquivalentChromaticIntervalClassVector.**__ne__**()
x.__ne__(y) <==> x!=y

InversionEquivalentChromaticIntervalClassVector.**__repr__**()

InversionEquivalentChromaticIntervalClassVector.**__setitem__**(**args*)

21.2.18 pitchtools.InversionEquivalentDiatonicIntervalClass



class pitchtools.**InversionEquivalentDiatonicIntervalClass**(**args*)

New in version 2.0. Abjad model of inversion-equivalent diatonic interval-class:

```
>>> pitchtools.InversionEquivalentDiatonicIntervalClass('-m14')
InversionEquivalentDiatonicIntervalClass('M2')
```

Inversion-equivalent diatonic interval-classes are immutable.

Read-only properties

InversionEquivalentDiatonicIntervalClass.**number**

InversionEquivalentDiatonicIntervalClass.**quality_string**

InversionEquivalentDiatonicIntervalClass.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

InversionEquivalentDiatonicIntervalClass.**__abs__**()

InversionEquivalentDiatonicIntervalClass.**__eq__**(*arg*)

InversionEquivalentDiatonicIntervalClass.**__float__**()

`InversionEquivalentDiatonicIntervalClass.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`InversionEquivalentDiatonicIntervalClass.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`InversionEquivalentDiatonicIntervalClass.__hash__()`

`InversionEquivalentDiatonicIntervalClass.__int__()`

`InversionEquivalentDiatonicIntervalClass.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`InversionEquivalentDiatonicIntervalClass.__lt__(expr)`
Abjad objects by default do not implement this method.

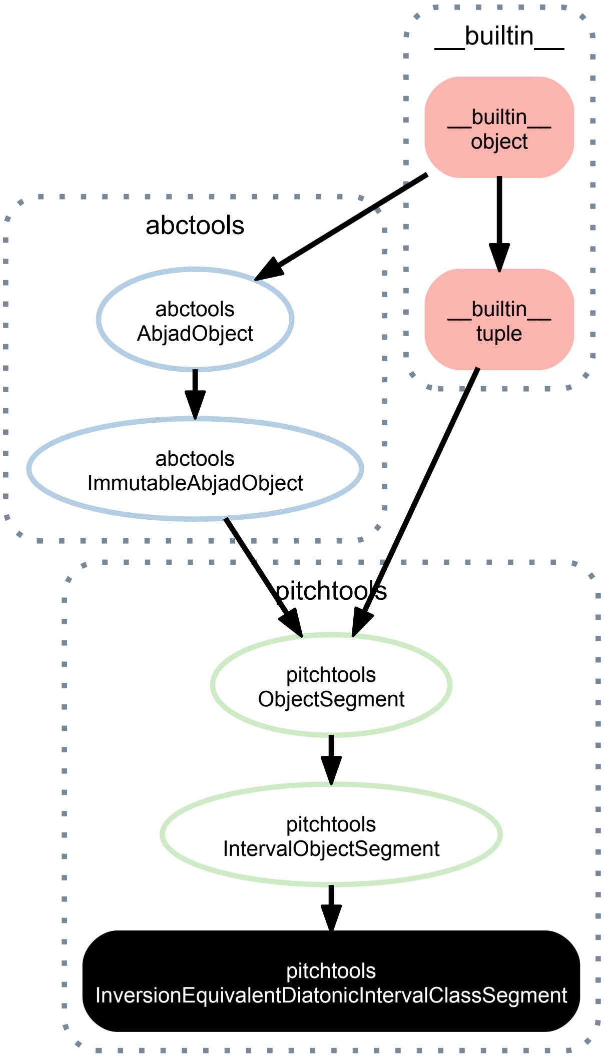
Raise exception.

`InversionEquivalentDiatonicIntervalClass.__ne__(arg)`

`InversionEquivalentDiatonicIntervalClass.__repr__()`

`InversionEquivalentDiatonicIntervalClass.__str__()`

21.2.19 pitchtools.InversionEquivalentDiatonicIntervalClassSegment



class `pitchtools.`**`InversionEquivalentDiatonicIntervalClassSegment`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of inversion-equivalent diatonic interval-class segment:

```
>>> pitchtools.InversionEquivalentDiatonicIntervalClassSegment(
... [('major', 2), ('major', 9), ('minor', -2), ('minor', -9)])
InversionEquivalentDiatonicIntervalClassSegment(M2, M2, m2, m2)
```

Inversion-equivalent diatonic interval-class segments are immutable.

Read-only properties

`InversionEquivalentDiatonicIntervalClassSegment.`**`interval_classes`**

`InversionEquivalentDiatonicIntervalClassSegment.`**`intervals`**

`InversionEquivalentDiatonicIntervalClassSegment.`**`is_tertian`**

True when all diatonic interval-classes in segment are tertian. Otherwise false:

```
>>> dics = pitchtools.InversionEquivalentDiatonicIntervalClassSegment(
... [('major', 3), ('minor', 6), ('major', 6)])
>>> dics.is_tertian
True
```

Return boolean.

`InversionEquivalentDiatonicIntervalClassSegment.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`InversionEquivalentDiatonicIntervalClassSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`InversionEquivalentDiatonicIntervalClassSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

`InversionEquivalentDiatonicIntervalClassSegment.`**`rotate`**(*n*)

Special methods

`InversionEquivalentDiatonicIntervalClassSegment.`**`__add__`**(*arg*)

`InversionEquivalentDiatonicIntervalClassSegment.`**`__contains__`**()
x.__contains__(y) <==> y in x

`InversionEquivalentDiatonicIntervalClassSegment.`**`__copy__`**()

`InversionEquivalentDiatonicIntervalClassSegment.`**`__eq__`**()
x.__eq__(y) <==> x==y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__ge__`**()
x.__ge__(y) <==> x>=y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__getitem__`**()
x.__getitem__(y) <==> x[y]

`InversionEquivalentDiatonicIntervalClassSegment.`**`__getslice__`**(*start*, *stop*)

`InversionEquivalentDiatonicIntervalClassSegment.`**`__gt__`**()
x.__gt__(y) <==> x>y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__hash__`**() <==> *hash(x)*

`InversionEquivalentDiatonicIntervalClassSegment.`**`__iter__`**() <==> *iter(x)*

`InversionEquivalentDiatonicIntervalClassSegment.`**`__le__`**()
x.__le__(y) <==> x<=y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__len__`**() <==> *len(x)*

`InversionEquivalentDiatonicIntervalClassSegment.`**`__lt__`**()
x.__lt__(y) <==> x<y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__mul__`**(*n*)

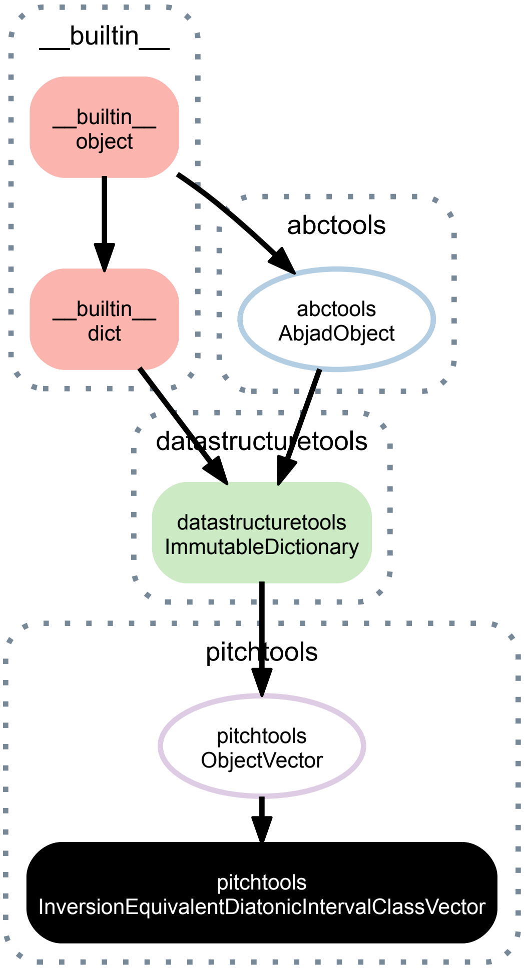
`InversionEquivalentDiatonicIntervalClassSegment.`**`__ne__`**()
x.__ne__(y) <==> x!=y

`InversionEquivalentDiatonicIntervalClassSegment.`**`__repr__`**()

`InversionEquivalentDiatonicIntervalClassSegment.`**`__rmul__`**(*n*)

`InversionEquivalentDiatonicIntervalClassSegment.`**`__str__`**()

21.2.20 pitchtools.InversionEquivalentDiatonicIntervalClassVector



class `pitchtools.`**`InversionEquivalentDiatonicIntervalClassVector`**(*expr*)

New in version 2.0. Abjad model of inversion-equivalent diatonic interval-class vector:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8")
>>> vector = pitchtools.InversionEquivalentDiatonicIntervalClassVector(staff)
```

```
>>> print vector
{P1: 0, aug1: 0, m2: 1, M2: 3, aug2: 0, dim3: 0, m3: 2, M3: 1, dim4: 0, P4: 3, aug4: 0}
```

Inversion-equivalent diatonic interval-class vector are not quatertone-aware.

Inversion-equivalent diatonic interval-class vectors are immutable.

Read-only properties

`InversionEquivalentDiatonicIntervalClassVector.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`InversionEquivalentDiatonicIntervalClassVector.`**`clear`**() → None. Remove all items from D.

`InversionEquivalentDiatonicIntervalClassVector.`**`copy`**() → a shallow copy of D

`InversionEquivalentDiatonicIntervalClassVector.`**get**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`InversionEquivalentDiatonicIntervalClassVector.`**has_key**(*k*) → True if D has a key k, else False

`InversionEquivalentDiatonicIntervalClassVector.`**items**() → list of D's (key, value) pairs, as 2-tuples

`InversionEquivalentDiatonicIntervalClassVector.`**iteritems**() → an iterator over the (key, value) items of D

`InversionEquivalentDiatonicIntervalClassVector.`**iterkeys**() → an iterator over the keys of D

`InversionEquivalentDiatonicIntervalClassVector.`**itervalues**() → an iterator over the values of D

`InversionEquivalentDiatonicIntervalClassVector.`**keys**() → list of D's keys

`InversionEquivalentDiatonicIntervalClassVector.`**pop**(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

`InversionEquivalentDiatonicIntervalClassVector.`**popitem**() → (k, v), remove and return some (key, value) pair as a

2-tuple; but raise KeyError if D is empty.

`InversionEquivalentDiatonicIntervalClassVector.`**setdefault**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

`InversionEquivalentDiatonicIntervalClassVector.`**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`InversionEquivalentDiatonicIntervalClassVector.`**values**() → list of D's values

`InversionEquivalentDiatonicIntervalClassVector.`**viewitems**() → a set-like object providing a view on D's items

`InversionEquivalentDiatonicIntervalClassVector.`**viewkeys**() → a set-like object providing a view on D's keys

`InversionEquivalentDiatonicIntervalClassVector.`**viewvalues**() → an object providing a view on D's values

Special methods

`InversionEquivalentDiatonicIntervalClassVector.`**__cmp__**(*y*) *<==> cmp(x, y)*

`InversionEquivalentDiatonicIntervalClassVector.`**__contains__**(*k*) → True if D has a key k, else False

`InversionEquivalentDiatonicIntervalClassVector.`**__delitem__**(**args*)

`InversionEquivalentDiatonicIntervalClassVector.`**__eq__**(*arg*)

`InversionEquivalentDiatonicIntervalClassVector.`**__ge__**()

x.__ge__(y) <==> x>=y

`InversionEquivalentDiatonicIntervalClassVector.__getitem__()`
x.__getitem__(y) <==> x[y]

`InversionEquivalentDiatonicIntervalClassVector.__gt__()`
x.__gt__(y) <==> x>y

`InversionEquivalentDiatonicIntervalClassVector.__iter__()` <==> *iter(x)*

`InversionEquivalentDiatonicIntervalClassVector.__le__()`
x.__le__(y) <==> x<=y

`InversionEquivalentDiatonicIntervalClassVector.__len__()` <==> *len(x)*

`InversionEquivalentDiatonicIntervalClassVector.__lt__()`
x.__lt__(y) <==> x<y

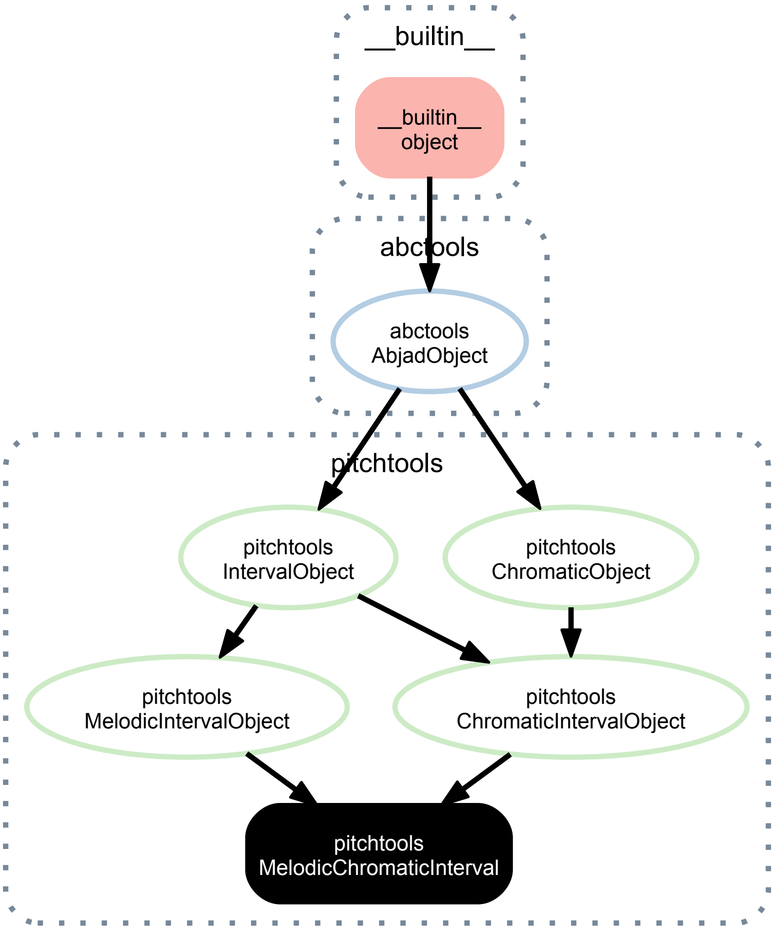
`InversionEquivalentDiatonicIntervalClassVector.__ne__(arg)`

`InversionEquivalentDiatonicIntervalClassVector.__repr__()`

`InversionEquivalentDiatonicIntervalClassVector.__setitem__(*args)`

`InversionEquivalentDiatonicIntervalClassVector.__str__()`

21.2.21 pitchtools.MelodicChromaticInterval



class `pitchtools.MelodicChromaticInterval(arg)`
New in version 2.0. Abjad model of melodic chromatic interval:

```
>>> pitchtools.MelodicChromaticInterval(-14)
MelodicChromaticInterval(-14)
```

Melodic chromatic intervals are immutable.

Read-only properties

MelodicChromaticInterval.**cents**

MelodicChromaticInterval.**chromatic_interval_number**
Read-only chromatic interval number:

```
>>> pitchtools.MelodicChromaticInterval(-14).chromatic_interval_number
-14
```

Return integer or float.

MelodicChromaticInterval.**direction_number**
Read-only numeric sign:

```
>>> pitchtools.MelodicChromaticInterval(-14).direction_number
-1
```

Return integer.

MelodicChromaticInterval.**direction_string**

MelodicChromaticInterval.**harmonic_chromatic_interval**
Read-only harmonic chromatic interval:

```
>>> pitchtools.MelodicChromaticInterval(-14).harmonic_chromatic_interval
HarmonicChromaticInterval(14)
```

Return harmonic chromatic interval.

MelodicChromaticInterval.**interval_class**

MelodicChromaticInterval.**melodic_chromatic_interval_class**
Read-only melodic chromatic interval-class:

```
>>> pitchtools.MelodicChromaticInterval(-14).melodic_chromatic_interval_class
MelodicChromaticIntervalClass(-2)
```

Return melodic chromatic interval-class.

MelodicChromaticInterval.**number**

MelodicChromaticInterval.**semitones**

MelodicChromaticInterval.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

MelodicChromaticInterval.**__abs__**()

MelodicChromaticInterval.**__add__**(*arg*)

MelodicChromaticInterval.**__copy__**()

MelodicChromaticInterval.**__eq__**(*arg*)

MelodicChromaticInterval.**__float__**()

MelodicChromaticInterval.**__ge__**(*arg*)

MelodicChromaticInterval.**__gt__**(*arg*)

MelodicChromaticInterval.**__hash__**()

MelodicChromaticInterval.**__int__**()

MelodicChromaticInterval.**__le__**(*arg*)

MelodicChromaticInterval.**__lt__**(*arg*)

MelodicChromaticInterval.**__ne__**(*arg*)

MelodicChromaticInterval.**__neg__**()

MelodicChromaticInterval.**__repr__**()

MelodicChromaticInterval.**__str__**()

MelodicChromaticInterval.**__sub__**(*arg*)

21.2.22 pitchtools.MelodicChromaticIntervalClass

class pitchtools.**MelodicChromaticIntervalClass**(*token*)

New in version 2.0. Abjad model of melodic chromatic interval-class:

```
>>> pitchtools.MelodicChromaticIntervalClass(-14)
MelodicChromaticIntervalClass(-2)
```

Melodic chromatic interval-classes are immutable.

Read-only properties

MelodicChromaticIntervalClass.**direction_number**

MelodicChromaticIntervalClass.**direction_symbol**

MelodicChromaticIntervalClass.**direction_word**

MelodicChromaticIntervalClass.**number**

MelodicChromaticIntervalClass.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

MelodicChromaticIntervalClass.**__abs__**()

MelodicChromaticIntervalClass.**__eq__**(*arg*)

MelodicChromaticIntervalClass.**__float__**()

MelodicChromaticIntervalClass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicChromaticIntervalClass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MelodicChromaticIntervalClass.**__hash__**()

MelodicChromaticIntervalClass.**__int__**()

MelodicChromaticIntervalClass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicChromaticIntervalClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

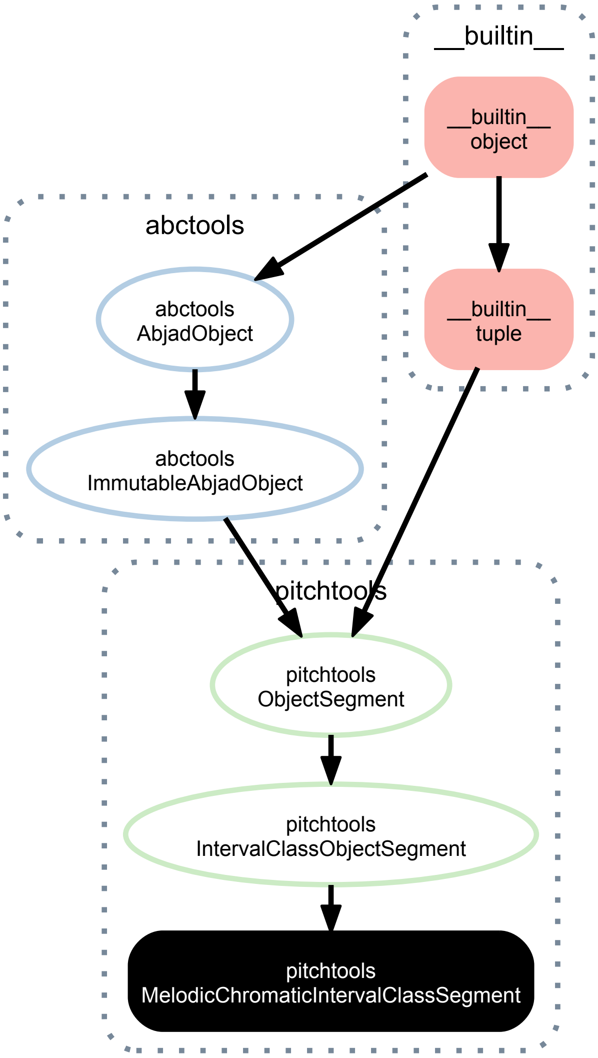
Raise exception.

MelodicChromaticIntervalClass.**__ne__**(*arg*)

MelodicChromaticIntervalClass.**__repr__**()

MelodicChromaticIntervalClass.**__str__**()

21.2.23 pitchtools.MelodicChromaticIntervalClassSegment



class `pitchtools.`**`MelodicChromaticIntervalClassSegment`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of melodic chromatic interval-class segment:

```
>>> pitchtools.MelodicChromaticIntervalClassSegment([-2, -14, 3, 5.5, 6.5])
MelodicChromaticIntervalClassSegment(-2, -2, +3, +5.5, +6.5)
```

Melodic chromatic interval-class segments are immutable.

Read-only properties

`MelodicChromaticIntervalClassSegment.`**`interval_class_numbers`**

`MelodicChromaticIntervalClassSegment.`**`interval_classes`**

`MelodicChromaticIntervalClassSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`MelodicChromaticIntervalClassSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`MelodicChromaticIntervalClassSegment.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.

Raises ValueError if the value is not present.

Special methods

`MelodicChromaticIntervalClassSegment.`**`__add__`**`(`*arg*`)`

`MelodicChromaticIntervalClassSegment.`**`__contains__`**`()`
x.__contains__(y) <==> y in x

`MelodicChromaticIntervalClassSegment.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`MelodicChromaticIntervalClassSegment.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`MelodicChromaticIntervalClassSegment.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`MelodicChromaticIntervalClassSegment.`**`__getslice__`**`(`*start*, *stop*`)`

`MelodicChromaticIntervalClassSegment.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`MelodicChromaticIntervalClassSegment.`**`__hash__`**`()` <==> *hash(x)*

`MelodicChromaticIntervalClassSegment.`**`__iter__`**`()` <==> *iter(x)*

`MelodicChromaticIntervalClassSegment.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`MelodicChromaticIntervalClassSegment.`**`__len__`**`()` <==> *len(x)*

`MelodicChromaticIntervalClassSegment.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

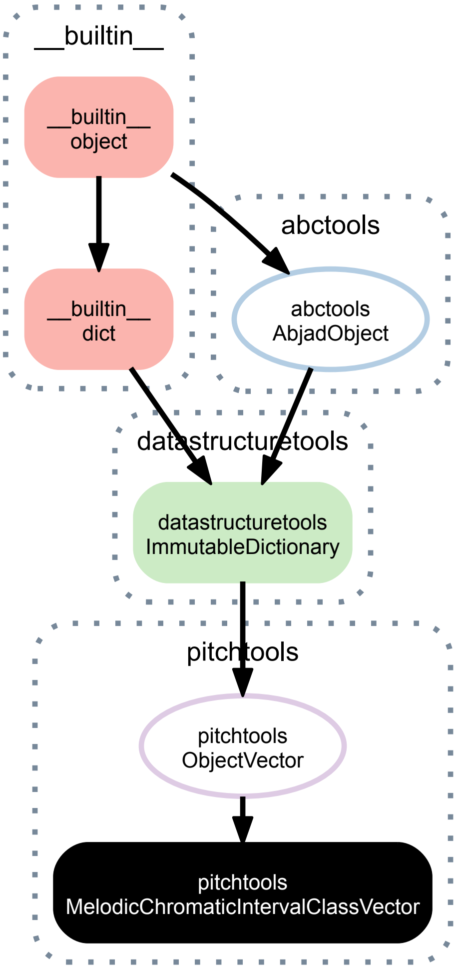
`MelodicChromaticIntervalClassSegment.`**`__mul__`**`(`*n*`)`

`MelodicChromaticIntervalClassSegment.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`MelodicChromaticIntervalClassSegment.`**`__repr__`**`()`

`MelodicChromaticIntervalClassSegment.`**`__rmul__`**`(`*n*`)`

21.2.24 pitchtools.MelodicChromaticIntervalClassVector



class `pitchtools.`**`MelodicChromaticIntervalClassVector`**(*mcic_tokens*)

New in version 2.0. Abjad model of melodic chromatic interval-class vector:

```
>>> print pitchtools.MelodicChromaticIntervalClassVector([-2, -14, 3, 5.5, 6.5])
    .   |   .   .   1   .   .   .   |   .   .   .   .   .   .
        |   .   2   .   .   .   .   |   .   .   .   .   .   .
        |   .   .   .   .   .   1   |   1   .   .   .   .   .
        |   .   .   .   .   .   .   |   .   .   .   .   .   .
```

Melodic chromatic interval-class vectors are immutable.

Read-only properties

`MelodicChromaticIntervalClassVector.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`MelodicChromaticIntervalClassVector.`**`clear`**() → None. Remove all items from D.

`MelodicChromaticIntervalClassVector.`**`copy`**() → a shallow copy of D

`MelodicChromaticIntervalClassVector.`**`get`**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`MelodicChromaticIntervalClassVector.`**`has_key`**`(k)` → True if D has a key k, else False

`MelodicChromaticIntervalClassVector.`**`items`**`()` → list of D's (key, value) pairs, as 2-tuples

`MelodicChromaticIntervalClassVector.`**`iteritems`**`()` → an iterator over the (key, value) items of D

`MelodicChromaticIntervalClassVector.`**`iterkeys`**`()` → an iterator over the keys of D

`MelodicChromaticIntervalClassVector.`**`itervalues`**`()` → an iterator over the values of D

`MelodicChromaticIntervalClassVector.`**`keys`**`()` → list of D's keys

`MelodicChromaticIntervalClassVector.`**`pop`**`(k[, d])` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`MelodicChromaticIntervalClassVector.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`MelodicChromaticIntervalClassVector.`**`setdefault`**`(k[, d])` → D.get(k,d), also set D[k]=d if k not in D

`MelodicChromaticIntervalClassVector.`**`update`**`([E], **F)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`MelodicChromaticIntervalClassVector.`**`values`**`()` → list of D's values

`MelodicChromaticIntervalClassVector.`**`viewitems`**`()` → a set-like object providing a view on D's items

`MelodicChromaticIntervalClassVector.`**`viewkeys`**`()` → a set-like object providing a view on D's keys

`MelodicChromaticIntervalClassVector.`**`viewvalues`**`()` → an object providing a view on D's values

Special methods

`MelodicChromaticIntervalClassVector.`**`__cmp__`**`(y)` <==> *cmp(x, y)*

`MelodicChromaticIntervalClassVector.`**`__contains__`**`(k)` → True if D has a key k, else False

`MelodicChromaticIntervalClassVector.`**`__delitem__`**`(*args)`

`MelodicChromaticIntervalClassVector.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`MelodicChromaticIntervalClassVector.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`MelodicChromaticIntervalClassVector.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`MelodicChromaticIntervalClassVector.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`MelodicChromaticIntervalClassVector.`**`__iter__`**`()` <==> *iter(x)*

`MelodicChromaticIntervalClassVector.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`MelodicChromaticIntervalClassVector.`**`__len__`**`()`

`MelodicChromaticIntervalClassVector.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

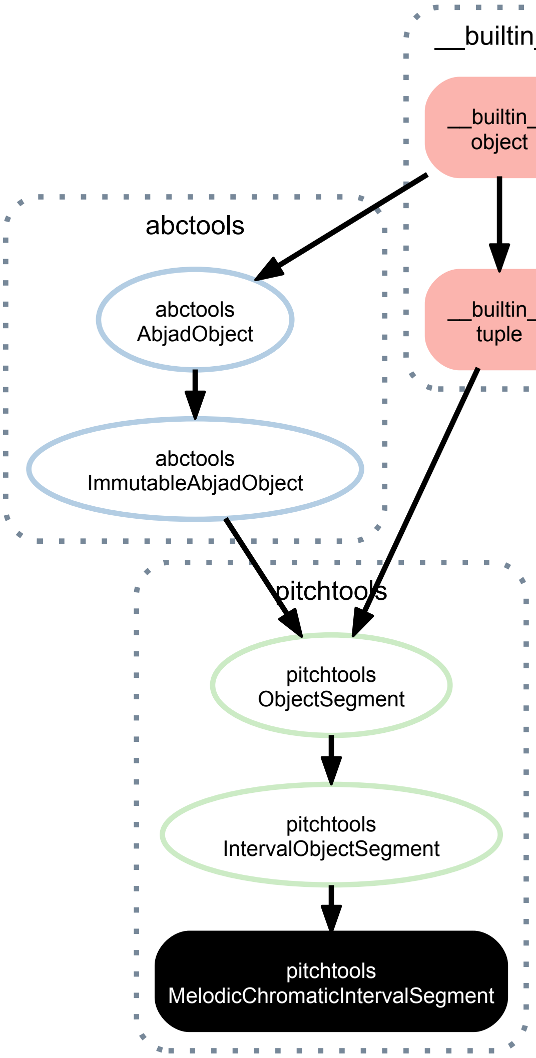
MelodicChromaticIntervalClassVector.**__ne__**()
x.__ne__(y) <==> x!=y

MelodicChromaticIntervalClassVector.**__repr__**()

MelodicChromaticIntervalClassVector.**__setitem__**(**args*)

MelodicChromaticIntervalClassVector.**__str__**()

21.2.25 pitchtools.MelodicChromaticIntervalSegment



class pitchtools.**MelodicChromaticIntervalSegment**(**args*, ***kwargs*)
New in version 2.0. Abjad model of melodic chromatic interval segment:

```
>>> pitchtools.MelodicChromaticIntervalSegment([11, 13, 13.5, -2, 2.5])
MelodicChromaticIntervalSegment(+11, +13, +13.5, -2, +2.5)
```

Melodic chromatic interval segments are immutable.

Read-only properties

MelodicChromaticIntervalSegment.**harmonic_chromatic_interval_segment**

MelodicChromaticIntervalSegment.**interval_classes**

MelodicChromaticIntervalSegment.**intervals**

`MelodicChromaticIntervalSegment.`**`melodic_chromatic_interval_class_segment`**

`MelodicChromaticIntervalSegment.`**`melodic_chromatic_interval_class_vector`**

`MelodicChromaticIntervalSegment.`**`melodic_chromatic_interval_numbers`**

`MelodicChromaticIntervalSegment.`**`slope`**

The slope of a melodic interval segment is the sum of its intervals divided by its length:

```
>>> pitchtools.MelodicChromaticIntervalSegment([1, 2]).slope
Fraction(3, 2)
```

Return fraction.

`MelodicChromaticIntervalSegment.`**`spread`**

The maximum harmonic interval spanned by any combination of the intervals within a harmonic chromatic interval segment:

```
>>> pitchtools.MelodicChromaticIntervalSegment([1, 2, -3, 1, -2, 1]).spread
HarmonicChromaticInterval(4)
>>> pitchtools.MelodicChromaticIntervalSegment([1, 1, 1, 2, -3, -2]).spread
HarmonicChromaticInterval(5)
```

Return harmonic chromatic interval.

`MelodicChromaticIntervalSegment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`MelodicChromaticIntervalSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`MelodicChromaticIntervalSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

`MelodicChromaticIntervalSegment.`**`rotate`**(*n*)

Special methods

`MelodicChromaticIntervalSegment.`**`__add__`**(*arg*)

`MelodicChromaticIntervalSegment.`**`__contains__`**()

x.__contains__(y) <==> y in x

`MelodicChromaticIntervalSegment.`**`__copy__`**()

`MelodicChromaticIntervalSegment.`**`__eq__`**()

x.__eq__(y) <==> x==y

`MelodicChromaticIntervalSegment.`**`__ge__`**()

x.__ge__(y) <==> x>=y

`MelodicChromaticIntervalSegment.`**`__getitem__`**()

x.__getitem__(y) <==> x[y]

`MelodicChromaticIntervalSegment.`**`__getslice__`**(*start*, *stop*)

`MelodicChromaticIntervalSegment.`**`__gt__`**()

x.__gt__(y) <==> x>y

`MelodicChromaticIntervalSegment.`**`__hash__`**() <==> *hash(x)*

`MelodicChromaticIntervalSegment.`**`__iter__`**() <==> *iter(x)*

MelodicChromaticIntervalSegment.**__le__**()
x.__le__(y) <==> x<=y

MelodicChromaticIntervalSegment.**__len__**() <==> *len(x)*

MelodicChromaticIntervalSegment.**__lt__**()
x.__lt__(y) <==> x<y

MelodicChromaticIntervalSegment.**__mul__**(*n*)

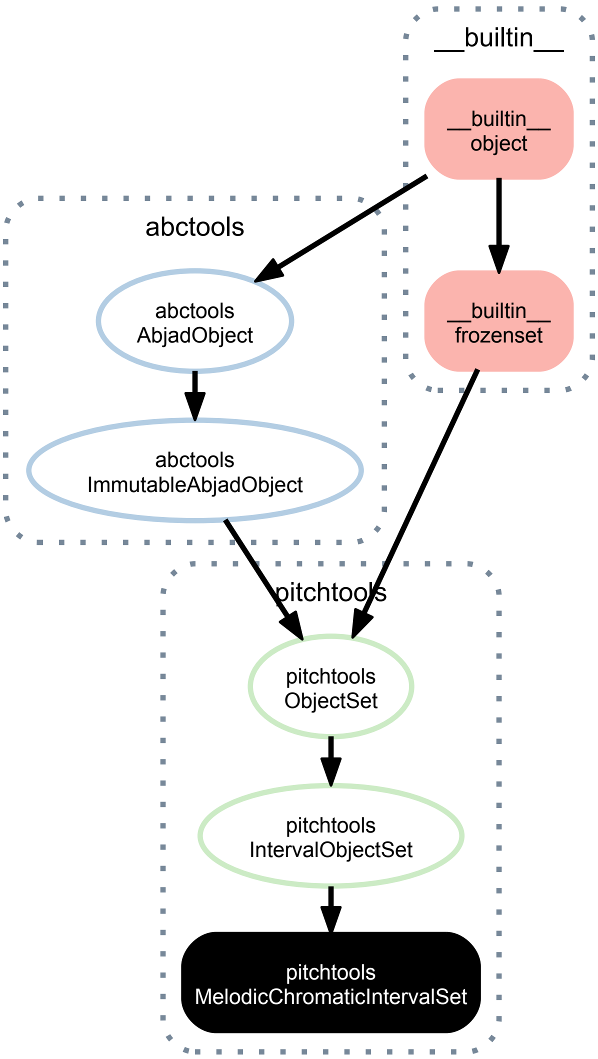
MelodicChromaticIntervalSegment.**__ne__**()
x.__ne__(y) <==> x!=y

MelodicChromaticIntervalSegment.**__repr__**()

MelodicChromaticIntervalSegment.**__rmul__**(*n*)

MelodicChromaticIntervalSegment.**__str__**()

21.2.26 pitchtools.MelodicChromaticIntervalSet



class pitchtools.**MelodicChromaticIntervalSet**(**args*, ***kwargs*)
New in version 2.0. Abjad model of melodic chromatic interval set:

```
>>> pitchtools.MelodicChromaticIntervalSet([11, 11, 13.5, 13.5])
MelodicChromaticIntervalSet(+11, +13.5)
```

Melodic chromatic interval sets are immutable.

Read-only properties

MelodicChromaticIntervalSet.**harmonic_chromatic_interval_set**

MelodicChromaticIntervalSet.**melodic_chromatic_interval_numbers**

MelodicChromaticIntervalSet.**melodic_chromatic_intervals**

MelodicChromaticIntervalSet.**storage_format**
Storage format of Abjad object.

Return string.

Methods

MelodicChromaticIntervalSet.**copy**()
Return a shallow copy of a set.

MelodicChromaticIntervalSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

MelodicChromaticIntervalSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

MelodicChromaticIntervalSet.**isdisjoint**()
Return True if two sets have a null intersection.

MelodicChromaticIntervalSet.**issubset**()
Report whether another set contains this set.

MelodicChromaticIntervalSet.**issuperset**()
Report whether this set contains another set.

MelodicChromaticIntervalSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

MelodicChromaticIntervalSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

MelodicChromaticIntervalSet.**__and__**()
x.__and__(y) <==> x&y

MelodicChromaticIntervalSet.**__cmp__**(*y*) <==> *cmp(x, y)*

MelodicChromaticIntervalSet.**__contains__**()
x.__contains__(y) <==> y in x.

MelodicChromaticIntervalSet.**__copy__**()

MelodicChromaticIntervalSet.**__eq__**()
x.__eq__(y) <==> x==y

MelodicChromaticIntervalSet.**__ge__**()
x.__ge__(y) <==> x>=y

MelodicChromaticIntervalSet.**__gt__**()
x.__gt__(y) <==> x>y

MelodicChromaticIntervalSet.**__hash__**() <==> *hash(x)*

MelodicChromaticIntervalSet.**__iter__**() <==> *iter(x)*

MelodicChromaticIntervalSet.**__le__**()
x.__le__(y) <==> x<=y

MelodicChromaticIntervalSet.**__len__**() <==> *len(x)*

MelodicChromaticIntervalSet.**__lt__**()
x.__lt__(y) <==> x<y

MelodicChromaticIntervalSet.**__ne__**()
x.__ne__(y) <==> x!=y

MelodicChromaticIntervalSet.**__or__**()
x.__or__(y) <==> x|y

MelodicChromaticIntervalSet.**__rand__**()
x.__rand__(y) <==> y&x

MelodicChromaticIntervalSet.**__repr__**()

MelodicChromaticIntervalSet.**__ror__**()
x.__ror__(y) <==> y|x

MelodicChromaticIntervalSet.**__rsub__**()
x.__rsub__(y) <==> y-x

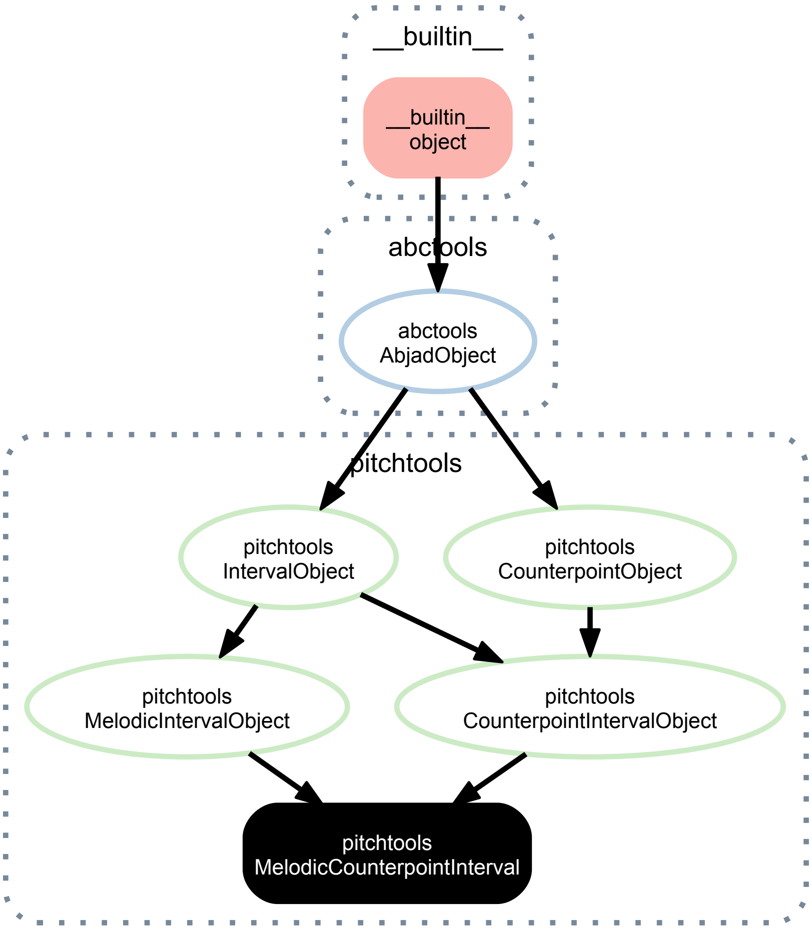
MelodicChromaticIntervalSet.**__rxor__**()
x.__rxor__(y) <==> y^x

MelodicChromaticIntervalSet.**__str__**()

MelodicChromaticIntervalSet.**__sub__**()
x.__sub__(y) <==> x-y

MelodicChromaticIntervalSet.**__xor__**()
x.__xor__(y) <==> x^y

21.2.27 pitchtools.MelodicCounterpointInterval



class `pitchtools.`**`MelodicCounterpointInterval`**(*number*)

New in version 2.0. Abjad model of melodic counterpoint interval:

```
>>> pitchtools.MelodicCounterpointInterval(-9)
MelodicCounterpointInterval(-9)
```

Melodic counterpoint intervals are immutable.

Read-only properties

`MelodicCounterpointInterval.`**`cents`**

`MelodicCounterpointInterval.`**`direction_number`**

`MelodicCounterpointInterval.`**`direction_string`**

`MelodicCounterpointInterval.`**`interval_class`**

`MelodicCounterpointInterval.`**`melodic_counterpoint_interval_class`**

`MelodicCounterpointInterval.`**`number`**

`MelodicCounterpointInterval.`**`semitones`**

`MelodicCounterpointInterval.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

MelodicCounterpointInterval.**__abs__**()

MelodicCounterpointInterval.**__eq__**(*arg*)

MelodicCounterpointInterval.**__float__**()

MelodicCounterpointInterval.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicCounterpointInterval.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MelodicCounterpointInterval.**__hash__**()

MelodicCounterpointInterval.**__int__**()

MelodicCounterpointInterval.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicCounterpointInterval.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicCounterpointInterval.**__ne__**(*arg*)

MelodicCounterpointInterval.**__neg__**()

MelodicCounterpointInterval.**__repr__**()

MelodicCounterpointInterval.**__str__**()

21.2.28 pitchtools.MelodicCounterpointIntervalClass

class `pitchtools.`**`MelodicCounterpointIntervalClass`**(*token*)

New in version 2.0. Abjad model of melodic counterpoint interval-class:

```
>>> pitchtools.MelodicCounterpointIntervalClass(-9)
MelodicCounterpointIntervalClass(-2)
```

Melodic counterpoint interval-classes are immutable.

Read-only properties

`MelodicCounterpointIntervalClass.`**`direction_number`**

`MelodicCounterpointIntervalClass.`**`direction_symbol`**

`MelodicCounterpointIntervalClass.`**`direction_word`**

`MelodicCounterpointIntervalClass.`**`number`**

`MelodicCounterpointIntervalClass.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`MelodicCounterpointIntervalClass.`**`__abs__`**()

`MelodicCounterpointIntervalClass.`**`__eq__`**(*arg*)

MelodicCounterpointIntervalClass.**__float__**()

MelodicCounterpointIntervalClass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicCounterpointIntervalClass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MelodicCounterpointIntervalClass.**__hash__**()

MelodicCounterpointIntervalClass.**__int__**()

MelodicCounterpointIntervalClass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MelodicCounterpointIntervalClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

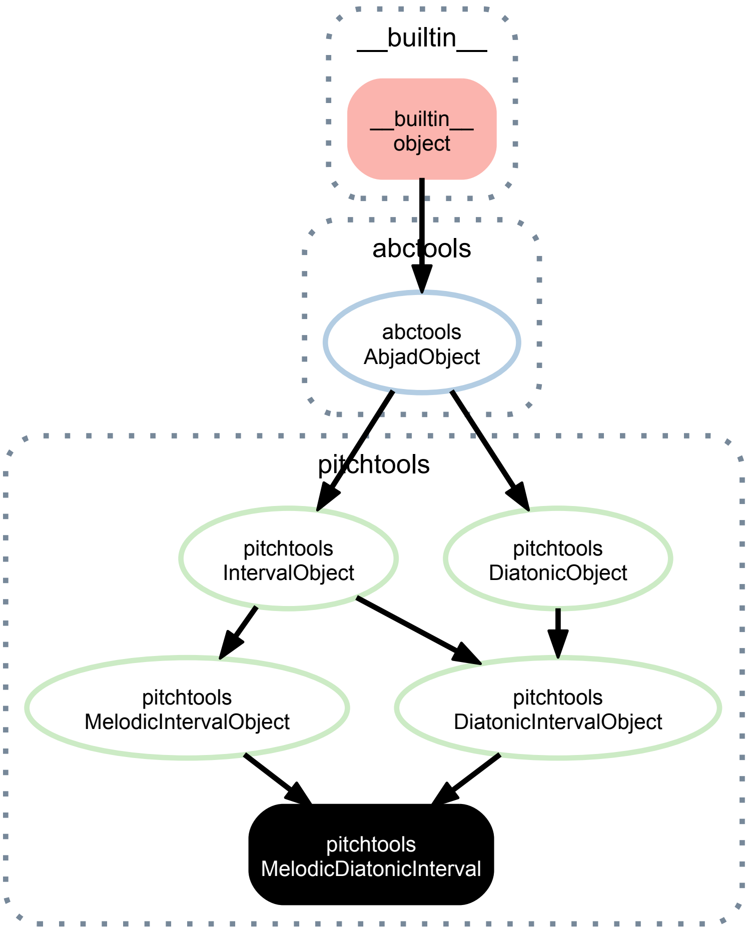
Raise exception.

MelodicCounterpointIntervalClass.**__ne__**(*arg*)

MelodicCounterpointIntervalClass.**__repr__**()

MelodicCounterpointIntervalClass.**__str__**()

21.2.29 pitchtools.MelodicDiatonicInterval



class `pitchtools.`**`MelodicDiatonicInterval`**`(`**args*`)`

New in version 2.0. Abjad model of melodic diatonic interval:

```
>>> pitchtools.MelodicDiatonicInterval('+M9')
MelodicDiatonicInterval('+M9')
```

Melodic diatonic intervals are immutable.

Read-only properties

`MelodicDiatonicInterval.`**`cents`**

`MelodicDiatonicInterval.`**`diatonic_interval_class`**

`MelodicDiatonicInterval.`**`direction_number`**

`MelodicDiatonicInterval.`**`direction_string`**

`MelodicDiatonicInterval.`**`harmonic_chromatic_interval`**

`MelodicDiatonicInterval.`**`harmonic_counterpoint_interval`**

`MelodicDiatonicInterval.`**`harmonic_diatonic_interval`**

`MelodicDiatonicInterval.`**`interval_class`**

`MelodicDiatonicInterval.`**`interval_string`**

`MelodicDiatonicInterval.`**`inversion_equivalent_chromatic_interval_class`**

`MelodicDiatonicInterval.`**`melodic_chromatic_interval`**

`MelodicDiatonicInterval.`**`melodic_counterpoint_interval`**

`MelodicDiatonicInterval.`**`melodic_diatonic_interval_class`**

`MelodicDiatonicInterval.`**`number`**

`MelodicDiatonicInterval.`**`quality_string`**

`MelodicDiatonicInterval.`**`semitones`**

`MelodicDiatonicInterval.`**`staff_spaces`**

`MelodicDiatonicInterval.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`MelodicDiatonicInterval.`**`__abs__`**`()`

`MelodicDiatonicInterval.`**`__add__`**`(`*arg*`)`

`MelodicDiatonicInterval.`**`__copy__`**`(`**args*`)`

`MelodicDiatonicInterval.`**`__deepcopy__`**`(`**args*`)`

`MelodicDiatonicInterval.`**`__eq__`**`(`*arg*`)`

`MelodicDiatonicInterval.`**`__float__`**`()`

`MelodicDiatonicInterval.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`MelodicDiatonicInterval.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`MelodicDiatonicInterval.__hash__()`

`MelodicDiatonicInterval.__int__()`

`MelodicDiatonicInterval.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`MelodicDiatonicInterval.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`MelodicDiatonicInterval.__mul__(arg)`

`MelodicDiatonicInterval.__ne__(arg)`

`MelodicDiatonicInterval.__neg__()`

`MelodicDiatonicInterval.__repr__()`

`MelodicDiatonicInterval.__rmul__(arg)`

`MelodicDiatonicInterval.__str__()`

`MelodicDiatonicInterval.__sub__(arg)`

21.2.30 pitchtools.MelodicDiatonicIntervalClass

class pitchtools.**MelodicDiatonicIntervalClass**(**args*)

New in version 2.0. Abjad model of melodic diatonic interval-class:

```
>>> pitchtools.MelodicDiatonicIntervalClass('-M9')
MelodicDiatonicIntervalClass('-M2')
```

Melodic diatonic interval-classes are immutable.

Read-only properties

MelodicDiatonicIntervalClass.**direction_number**

MelodicDiatonicIntervalClass.**direction_symbol**

MelodicDiatonicIntervalClass.**direction_word**

MelodicDiatonicIntervalClass.**number**

MelodicDiatonicIntervalClass.**quality_string**

MelodicDiatonicIntervalClass.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

MelodicDiatonicIntervalClass.**__abs__**()

MelodicDiatonicIntervalClass.**__eq__**(*arg*)

MelodicDiatonicIntervalClass.**__float__**()

MelodicDiatonicIntervalClass.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

MelodicDiatonicIntervalClass.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

MelodicDiatonicIntervalClass.**__hash__**()

MelodicDiatonicIntervalClass.**__int__**()

MelodicDiatonicIntervalClass.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

MelodicDiatonicIntervalClass.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

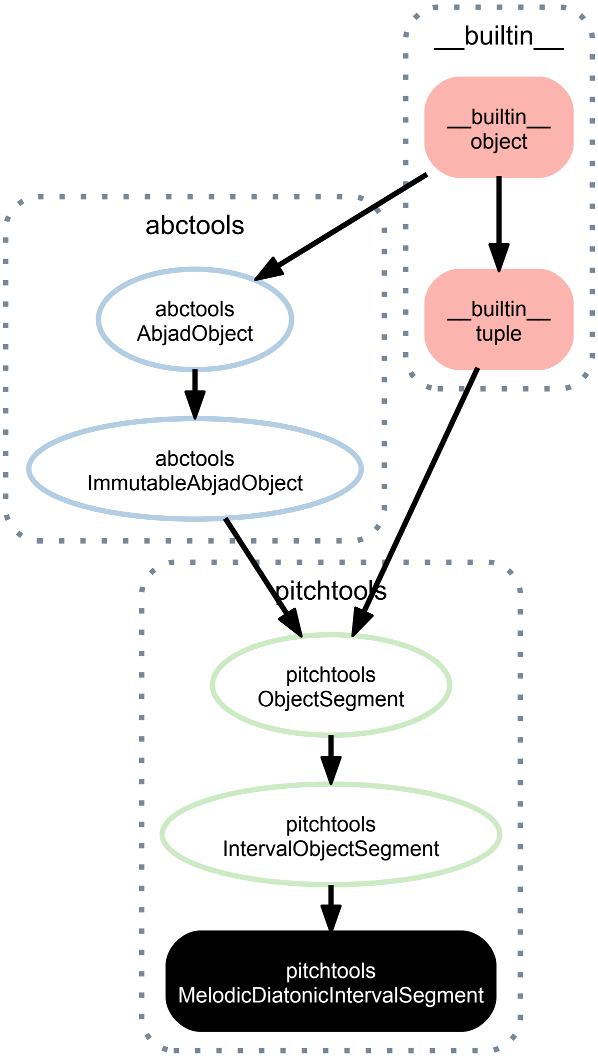
Raise exception.

MelodicDiatonicIntervalClass.**__ne__**(*arg*)

MelodicDiatonicIntervalClass.**__repr__**()

MelodicDiatonicIntervalClass.**__str__**()

21.2.31 pitchtools.MelodicDiatonicIntervalSegment



class pitchtools.**MelodicDiatonicIntervalSegment** (**args, **kwargs*)

New in version 2.0. Abjad model of melodic diatonic interval segment:

```
>>> pitchtools.MelodicDiatonicIntervalSegment('M2 M9 -m3 -P4')
MelodicDiatonicIntervalSegment('+M2 +M9 -m3 -P4')
```

Melodic diatonic interval segments are immutable.

Read-only properties

MelodicDiatonicIntervalSegment.**harmonic_chromatic_interval_segment**

MelodicDiatonicIntervalSegment.**harmonic_diatonic_interval_segment**

MelodicDiatonicIntervalSegment.**interval_classes**

MelodicDiatonicIntervalSegment.**intervals**

MelodicDiatonicIntervalSegment.**melodic_chromatic_interval_segment**

MelodicDiatonicIntervalSegment.**storage_format**

Storage format of Abjad object.

Return string.

Methods

`MelodicDiatonicIntervalSegment.`**`count`**(*value*) → integer – return number of occurrences of value

`MelodicDiatonicIntervalSegment.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

`MelodicDiatonicIntervalSegment.`**`rotate`**(*n*)

Special methods

`MelodicDiatonicIntervalSegment.`**`__add__`**(*arg*)

`MelodicDiatonicIntervalSegment.`**`__contains__`**()

x.__contains__(y) <==> y in x

`MelodicDiatonicIntervalSegment.`**`__copy__`**()

`MelodicDiatonicIntervalSegment.`**`__eq__`**()

x.__eq__(y) <==> x==y

`MelodicDiatonicIntervalSegment.`**`__ge__`**()

x.__ge__(y) <==> x>=y

`MelodicDiatonicIntervalSegment.`**`__getitem__`**()

x.__getitem__(y) <==> x[y]

`MelodicDiatonicIntervalSegment.`**`__getslice__`**(*start*, *stop*)

`MelodicDiatonicIntervalSegment.`**`__gt__`**()

x.__gt__(y) <==> x>y

`MelodicDiatonicIntervalSegment.`**`__hash__`**() <==> *hash(x)*

`MelodicDiatonicIntervalSegment.`**`__iter__`**() <==> *iter(x)*

`MelodicDiatonicIntervalSegment.`**`__le__`**()

x.__le__(y) <==> x<=y

`MelodicDiatonicIntervalSegment.`**`__len__`**() <==> *len(x)*

`MelodicDiatonicIntervalSegment.`**`__lt__`**()

x.__lt__(y) <==> x<y

`MelodicDiatonicIntervalSegment.`**`__mul__`**(*n*)

`MelodicDiatonicIntervalSegment.`**`__ne__`**()

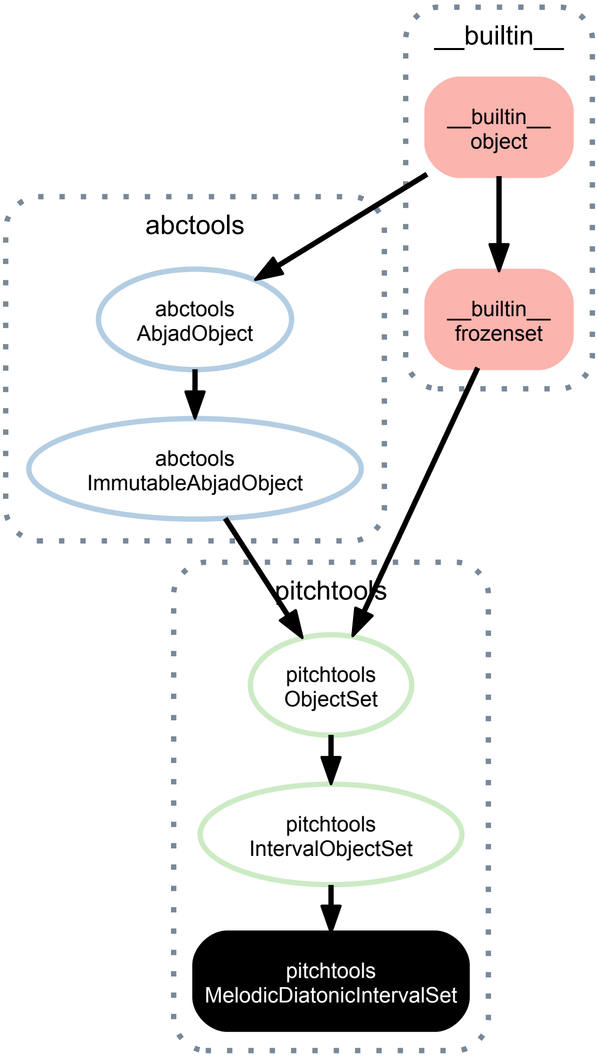
x.__ne__(y) <==> x!=y

`MelodicDiatonicIntervalSegment.`**`__repr__`**()

`MelodicDiatonicIntervalSegment.`**`__rmul__`**(*n*)

`MelodicDiatonicIntervalSegment.`**`__str__`**()

21.2.32 pitchtools.MelodicDiatonicIntervalSet



class `pitchtools.`**`MelodicDiatonicIntervalSet`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of melodic diatonic interval set:

```
>>> pitchtools.MelodicDiatonicIntervalSet('M2 M2 -m3 -P4')
MelodicDiatonicIntervalSet('-P4 -m3 +M2')
```

Melodic diatonic interval sets are immutable.

Read-only properties

`MelodicDiatonicIntervalSet.`**`harmonic_chromatic_interval_set`**

`MelodicDiatonicIntervalSet.`**`harmonic_diatonic_interval_set`**

`MelodicDiatonicIntervalSet.`**`melodic_chromatic_interval_set`**

`MelodicDiatonicIntervalSet.`**`melodic_diatonic_interval_numbers`**

`MelodicDiatonicIntervalSet.`**`melodic_diatonic_intervals`**

`MelodicDiatonicIntervalSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`MelodicDiatonicIntervalSet.`**`copy`**`()`
Return a shallow copy of a set.

`MelodicDiatonicIntervalSet.`**`difference`**`()`
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`MelodicDiatonicIntervalSet.`**`intersection`**`()`
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`MelodicDiatonicIntervalSet.`**`isdisjoint`**`()`
Return True if two sets have a null intersection.

`MelodicDiatonicIntervalSet.`**`issubset`**`()`
Report whether another set contains this set.

`MelodicDiatonicIntervalSet.`**`issuperset`**`()`
Report whether this set contains another set.

`MelodicDiatonicIntervalSet.`**`symmetric_difference`**`()`
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`MelodicDiatonicIntervalSet.`**`union`**`()`
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`MelodicDiatonicIntervalSet.`**`__and__`**`()`
x.__and__(y) <==> x&y

`MelodicDiatonicIntervalSet.`**`__cmp__`**`(y) <==> cmp(x, y)`

`MelodicDiatonicIntervalSet.`**`__contains__`**`()`
x.__contains__(y) <==> y in x.

`MelodicDiatonicIntervalSet.`**`__copy__`**`()`

`MelodicDiatonicIntervalSet.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`MelodicDiatonicIntervalSet.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`MelodicDiatonicIntervalSet.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`MelodicDiatonicIntervalSet.`**`__hash__`**`() <==> hash(x)`

`MelodicDiatonicIntervalSet.`**`__iter__`**`() <==> iter(x)`

`MelodicDiatonicIntervalSet.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`MelodicDiatonicIntervalSet.`**`__len__`**`() <==> len(x)`

`MelodicDiatonicIntervalSet.`**`__lt__`**`()`
x.__lt__(y) <==> x < y

`MelodicDiatonicIntervalSet.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`MelodicDiatonicIntervalSet.__or__()`
x.__or__(y) <==> x|y

`MelodicDiatonicIntervalSet.__rand__()`
x.__rand__(y) <==> y&x

`MelodicDiatonicIntervalSet.__repr__()`

`MelodicDiatonicIntervalSet.__ror__()`
x.__ror__(y) <==> y|x

`MelodicDiatonicIntervalSet.__rsub__()`
x.__rsub__(y) <==> y-x

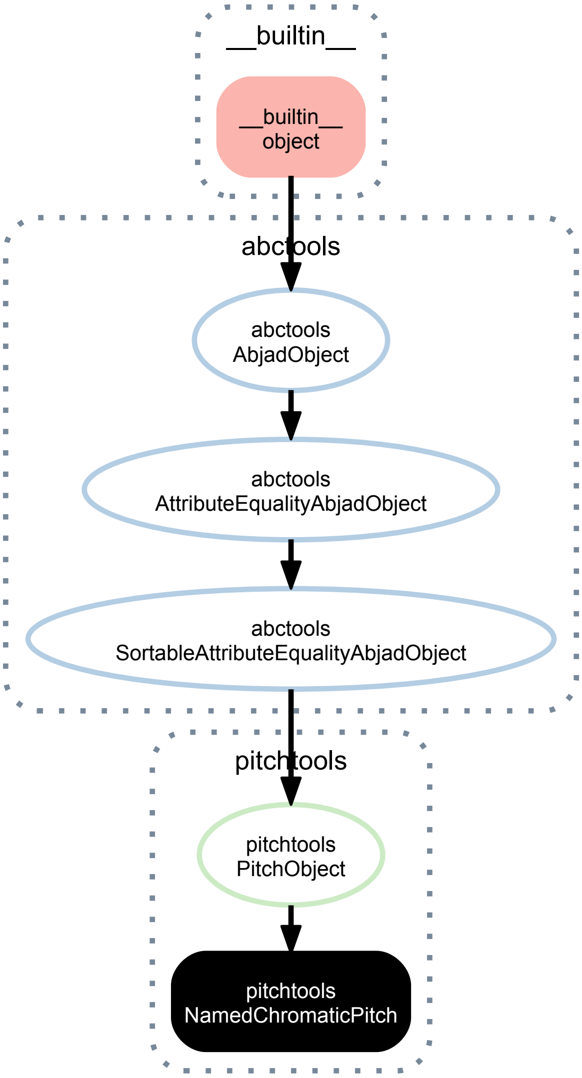
`MelodicDiatonicIntervalSet.__rxor__()`
x.__rxor__(y) <==> y^x

`MelodicDiatonicIntervalSet.__str__()`

`MelodicDiatonicIntervalSet.__sub__()`
x.__sub__(y) <==> x-y

`MelodicDiatonicIntervalSet.__xor__()`
x.__xor__(y) <==> x^y

21.2.33 pitchtools.NamedChromaticPitch



class pitchtools.**NamedChromaticPitch**(**args*, ***kwargs*)

New in version 1.1. Abjad model of named chromatic pitch:

```
>>> pitchtools.NamedChromaticPitch("cs''")
NamedChromaticPitch("cs''")
```

Named chromatic pitches are immutable.

Read-only properties

NamedChromaticPitch.**accidental_spelling**

Read-only accidental spelling:

```
>>> pitchtools.NamedChromaticPitch("c").accidental_spelling
'mixed'
```

Return string.

NamedChromaticPitch.**chromatic_pitch_class_name**

Read-only chromatic pitch-class name:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.chromatic_pitch_class_name
'cs'
```

Return string.

NamedChromaticPitch.**chromatic_pitch_class_number**

Read-only chromatic pitch-class number:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.chromatic_pitch_class_number
1
```

Return integer or float.

NamedChromaticPitch.**chromatic_pitch_name**

Read-only chromatic pitch name:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.chromatic_pitch_name
"cs''"
```

Return string.

NamedChromaticPitch.**chromatic_pitch_number**

Read-only chromatic pitch-class number:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.chromatic_pitch_number
13
```

Return integer or float.

NamedChromaticPitch.**deviation_in_cents**

Read-only deviation of named chromatic pitch in cents:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.deviation_in_cents is None
True
```

Return integer or none.

NamedChromaticPitch.**diatonic_pitch_class_name**

Read-only diatonic pitch-class name:

```
>>> named_diatonic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_diatonic_pitch.diatonic_pitch_class_name
'c'
```

Return string.

NamedChromaticPitch.**diatonic_pitch_class_number**

Read-only diatonic pitch-class number:

```
>>> named_diatonic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_diatonic_pitch.diatonic_pitch_class_number
0
```

Return integer.

NamedChromaticPitch.**diatonic_pitch_name**

Read-only diatonic pitch name:

```
>>> named_diatonic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_diatonic_pitch.diatonic_pitch_name
"c''"
```

Return string.

NamedChromaticPitch.**diatonic_pitch_number**

Read-only diatonic pitch number:

```
>>> named_diatonic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_diatonic_pitch.diatonic_pitch_number
7
```

Return integer.

NamedChromaticPitch.**lilypond_format**

Read-only LilyPond input format of named chromatic pitch:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.lilypond_format
"cs''"
```

Return string.

NamedChromaticPitch.**named_chromatic_pitch_class**

Read-only named pitch-class:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.named_chromatic_pitch_class
NamedChromaticPitchClass('cs')
```

Return named chromatic pitch-class.

NamedChromaticPitch.**named_diatonic_pitch**

Read-only named diatonic pitch:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.named_diatonic_pitch
NamedDiatonicPitch("c''")
```

Return named diatonic pitch.

NamedChromaticPitch.**named_diatonic_pitch_class**

Read-only named diatonic pitch-class:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.named_diatonic_pitch_class
NamedDiatonicPitchClass('c')
```

Return named diatonic pitch-class.

NamedChromaticPitch.**numbered_chromatic_pitch**

Read-only numbered chromatic pitch from named chromatic pitch:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.numbered_chromatic_pitch_class
NumberedChromaticPitchClass(1)
```

Return numbered chromatic pitch-class.

NamedChromaticPitch.**numbered_chromatic_pitch_class**
Read-only numbered pitch-class:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.numbered_chromatic_pitch_class
NumberedChromaticPitchClass(1)
```

Return numbered chromatic pitch-class.

NamedChromaticPitch.**numbered_diatonic_pitch**
Read-only numbered diatonic pitch:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.numbered_diatonic_pitch
NumberedDiatonicPitch(7)
```

Return numbered diatonic pitch.

NamedChromaticPitch.**numbered_diatonic_pitch_class**
Read-only numbered diatonic pitch:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.numbered_diatonic_pitch_class
NumberedDiatonicPitchClass(0)
```

Return numbered diatonic pitch-class.

NamedChromaticPitch.**octave_number**
Read-only integer octave number:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.octave_number
5
```

Return integer.

NamedChromaticPitch.**pitch_class_octave_label**
Read-only pitch-class / octave label:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
>>> named_chromatic_pitch.pitch_class_octave_label
'C#5'
```

Return string.

NamedChromaticPitch.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

NamedChromaticPitch.**__abs__**()

NamedChromaticPitch.**__add__**(*melodic_interval*)
New in version 2.0.

NamedChromaticPitch.**__copy__**(**args*)
New in version 2.0.

NamedChromaticPitch.**__deepcopy__**(**args*)
New in version 2.0.

NamedChromaticPitch.**__eq__**(*arg*)

NamedChromaticPitch.**__float__**()

NamedChromaticPitch.**__ge__**(*arg*)

NamedChromaticPitch.**__gt__**(*arg*)

NamedChromaticPitch.**__hash__**()

NamedChromaticPitch.**__int__**()

NamedChromaticPitch.**__le__**(*arg*)

NamedChromaticPitch.**__lt__**(*arg*)

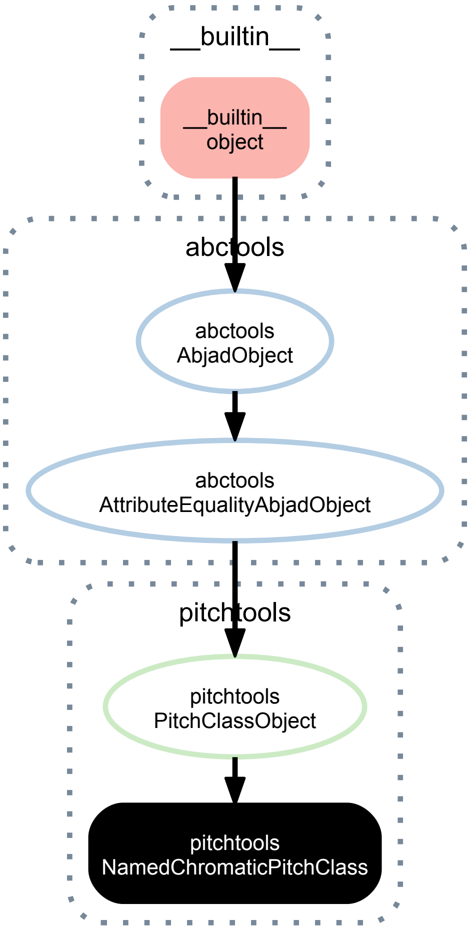
NamedChromaticPitch.**__ne__**(*arg*)

NamedChromaticPitch.**__repr__**()

NamedChromaticPitch.**__str__**()

NamedChromaticPitch.**__sub__**(*arg*)

21.2.34 pitchtools.NamedChromaticPitchClass



class pitchtools.**NamedChromaticPitchClass**(*arg*)

New in version 2.0. Abjad model of named chromatic pitch-class:

```
>>> ncpc = pitchtools.NamedChromaticPitchClass('cs')
```

```
>>> ncpc
NamedChromaticPitchClass('cs')
```

Named chromatic pitch-classes are immutable.

Read-only properties

`NamedChromaticPitchClass.`**numbered_chromatic_pitch_class**

Read-only numbered chromatic pitch-class:

```
>>> ncpc.numbered_chromatic_pitch_class
NumberedChromaticPitchClass(1)
```

Return numbered chromatic pitch-class.

`NamedChromaticPitchClass.`**storage_format**

Storage format of Abjad object.

Return string.

Methods

`NamedChromaticPitchClass.`**apply_accidental**(*accidental*)

Apply *accidental*:

```
>>> ncpc.apply_accidental('qs')
NamedChromaticPitchClass('ctqs')
```

Return named chromatic pitch-class.

`NamedChromaticPitchClass.`**transpose**(*melodic_diatonic_interval*)

Transpose named chromatic pitch-class by *melodic_diatonic_interval*:

```
>>> ncpc.transpose(pitchtools.MelodicDiatonicInterval('major', 2))
NamedChromaticPitchClass('ds')
```

Return named chromatic pitch-class.

Special methods

`NamedChromaticPitchClass.`**__abs__**()

`NamedChromaticPitchClass.`**__add__**(*melodic_diatonic_interval*)

`NamedChromaticPitchClass.`**__copy__**(**args*)

`NamedChromaticPitchClass.`**__deepcopy__**(**args*)

`NamedChromaticPitchClass.`**__eq__**(*arg*)

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`NamedChromaticPitchClass.`**__float__**()

`NamedChromaticPitchClass.`**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`NamedChromaticPitchClass.`**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`NamedChromaticPitchClass.`**__hash__**()

`NamedChromaticPitchClass.`**__int__**()

`NamedChromaticPitchClass.`**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

NamedChromaticPitchClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NamedChromaticPitchClass.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

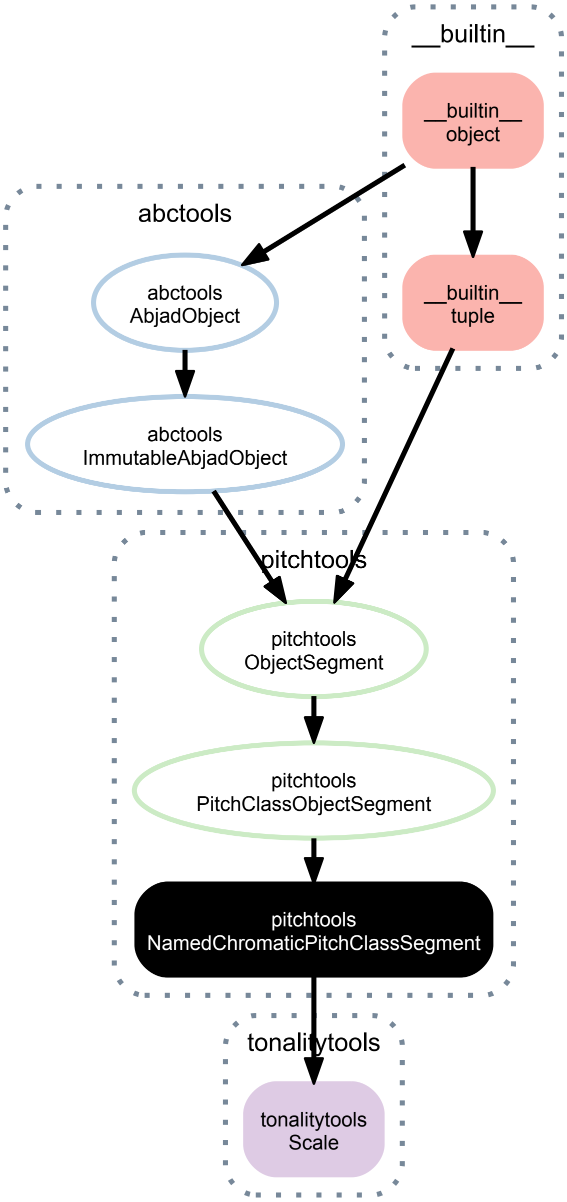
Return boolean.

NamedChromaticPitchClass.**__repr__**()

NamedChromaticPitchClass.**__str__**()

NamedChromaticPitchClass.**__sub__**(*arg*)

21.2.35 pitchtools.NamedChromaticPitchClassSegment



class pitchtools.**NamedChromaticPitchClassSegment**(**args*, ***kwargs*)
New in version 2.0. Abjad model of named chromatic pitch-class segment:

```
>>> pitchtools.NamedChromaticPitchClassSegment(['gs', 'a', 'as', 'c', 'cs'])
NamedChromaticPitchClassSegment(['gs', 'a', 'as', 'c', 'cs'])
```

Named chromatic pitch-class segments are immutable.

Read-only properties

NamedChromaticPitchClassSegment.**inversion_equivalent_diatonic_interval_class_segment**

NamedChromaticPitchClassSegment.**named_chromatic_pitch_class_set**

NamedChromaticPitchClassSegment.**named_chromatic_pitch_classes**

NamedChromaticPitchClassSegment.**numbered_chromatic_pitch_class_segment**

NamedChromaticPitchClassSegment.**numbered_chromatic_pitch_class_set**

NamedChromaticPitchClassSegment.**numbered_chromatic_pitch_classes**

NamedChromaticPitchClassSegment.**storage_format**
Storage format of Abjad object.

Return string.

Methods

NamedChromaticPitchClassSegment.**count**(*value*) → integer – return number of occurrences of value

NamedChromaticPitchClassSegment.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

NamedChromaticPitchClassSegment.**is_equivalent_under_transposition**(*arg*)

NamedChromaticPitchClassSegment.**retrograde**()

NamedChromaticPitchClassSegment.**rotate**(*n*)

NamedChromaticPitchClassSegment.**transpose**(*melodic_diatonic_interval*)

Special methods

NamedChromaticPitchClassSegment.**__add__**(*arg*)

NamedChromaticPitchClassSegment.**__contains__**()
x.__contains__(y) <==> y in x

NamedChromaticPitchClassSegment.**__eq__**()
x.__eq__(y) <==> x==y

NamedChromaticPitchClassSegment.**__ge__**()
x.__ge__(y) <==> x>=y

NamedChromaticPitchClassSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

NamedChromaticPitchClassSegment.**__getslice__**(*start*, *stop*)

NamedChromaticPitchClassSegment.**__gt__**()
x.__gt__(y) <==> x>y

NamedChromaticPitchClassSegment.**__hash__**() <==> *hash(x)*

NamedChromaticPitchClassSegment.**__iter__**() <==> *iter(x)*

NamedChromaticPitchClassSegment.**__le__**()
x.__le__(y) <==> x<=y

NamedChromaticPitchClassSegment.**__len__**() <==> *len(x)*

NamedChromaticPitchClassSegment.**__lt__**()
x.__lt__(y) <==> x<y

NamedChromaticPitchClassSegment.**__mul__**(*n*)

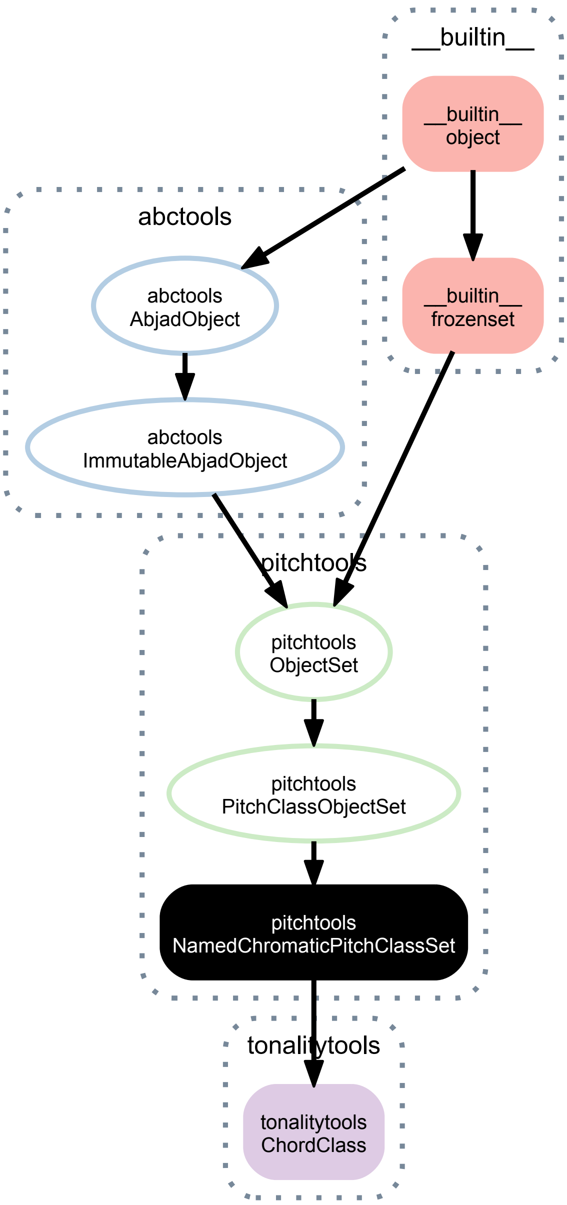
NamedChromaticPitchClassSegment.**__ne__**()
x.__ne__(y) <==> x!=y

NamedChromaticPitchClassSegment.**__repr__**()

NamedChromaticPitchClassSegment.**__rmul__**(*n*)

NamedChromaticPitchClassSegment.**__str__**()

21.2.36 pitchtools.NamedChromaticPitchClassSet



class `pitchtools.`**`NamedChromaticPitchClassSet`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of a named chromatic pitch-class set:

```
>>> named_chromatic_pitch_class_set = pitchtools.NamedChromaticPitchClassSet(
...     ['gs', 'g', 'as', 'c', 'cs'])
```

```
>>> named_chromatic_pitch_class_set
NamedChromaticPitchClassSet(['as', 'c', 'cs', 'g', 'gs'])
```

```
>>> print named_chromatic_pitch_class_set
{as, c, cs, g, gs}
```

Named chromatic pitch-class sets are immutable.

Read-only properties

`NamedChromaticPitchClassSet.`**`inversion_equivalent_diatonic_interval_class_vector`**

`NamedChromaticPitchClassSet.`**`named_chromatic_pitch_classes`**

Read-only named chromatic pitch-classes:

```
>>> named_chromatic_pitch_class_set = pitchtools.NamedChromaticPitchClassSet(
...     ['gs', 'g', 'as', 'c', 'cs'])
>>> for x in named_chromatic_pitch_class_set.named_chromatic_pitch_classes: x
...
NamedChromaticPitchClass('as')
NamedChromaticPitchClass('c')
NamedChromaticPitchClass('cs')
NamedChromaticPitchClass('g')
NamedChromaticPitchClass('gs')
```

Return tuple.

`NamedChromaticPitchClassSet.`**`numbered_chromatic_pitch_class_set`**

`NamedChromaticPitchClassSet.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`NamedChromaticPitchClassSet.`**`copy`**()

Return a shallow copy of a set.

`NamedChromaticPitchClassSet.`**`difference`**()

Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

`NamedChromaticPitchClassSet.`**`intersection`**()

Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

`NamedChromaticPitchClassSet.`**`isdisjoint`**()

Return True if two sets have a null intersection.

`NamedChromaticPitchClassSet.`**`issubset`**()

Report whether another set contains this set.

`NamedChromaticPitchClassSet.`**`issuperset`**()

Report whether this set contains another set.

`NamedChromaticPitchClassSet.`**`order_by`**(*npc_seg*)

NamedChromaticPitchClassSet.**symmetric_difference**()
    Return the symmetric difference of two sets as a new set.

    (i.e. all elements that are in exactly one of the sets.)

NamedChromaticPitchClassSet.**transpose**(*melodic_diatonic_interval*)
    Transpose all npcs in self by melodic diatonic interval.

NamedChromaticPitchClassSet.**union**()
    Return the union of sets as a new set.

    (i.e. all elements that are in either set.)

Special methods

NamedChromaticPitchClassSet.**__and__**()
    x.__and__(y) <==> x&y

NamedChromaticPitchClassSet.**__cmp__**(*y) <==> cmp(x, y*)

NamedChromaticPitchClassSet.**__contains__**()
    x.__contains__(y) <==> y in x.

NamedChromaticPitchClassSet.**__eq__**(*arg*)

NamedChromaticPitchClassSet.**__ge__**()
    x.__ge__(y) <==> x>=y

NamedChromaticPitchClassSet.**__gt__**()
    x.__gt__(y) <==> x>y

NamedChromaticPitchClassSet.**__hash__**()

NamedChromaticPitchClassSet.**__iter__**(*) <==> iter(x*)

NamedChromaticPitchClassSet.**__le__**()
    x.__le__(y) <==> x<=y

NamedChromaticPitchClassSet.**__len__**(*) <==> len(x*)

NamedChromaticPitchClassSet.**__lt__**()
    x.__lt__(y) <==> x<y

NamedChromaticPitchClassSet.**__ne__**(*arg*)

NamedChromaticPitchClassSet.**__or__**()
    x.__or__(y) <==> x|y

NamedChromaticPitchClassSet.**__rand__**()
    x.__rand__(y) <==> y&x

NamedChromaticPitchClassSet.**__repr__**()

NamedChromaticPitchClassSet.**__ror__**()
    x.__ror__(y) <==> y|x

NamedChromaticPitchClassSet.**__rsub__**()
    x.__rsub__(y) <==> y-x

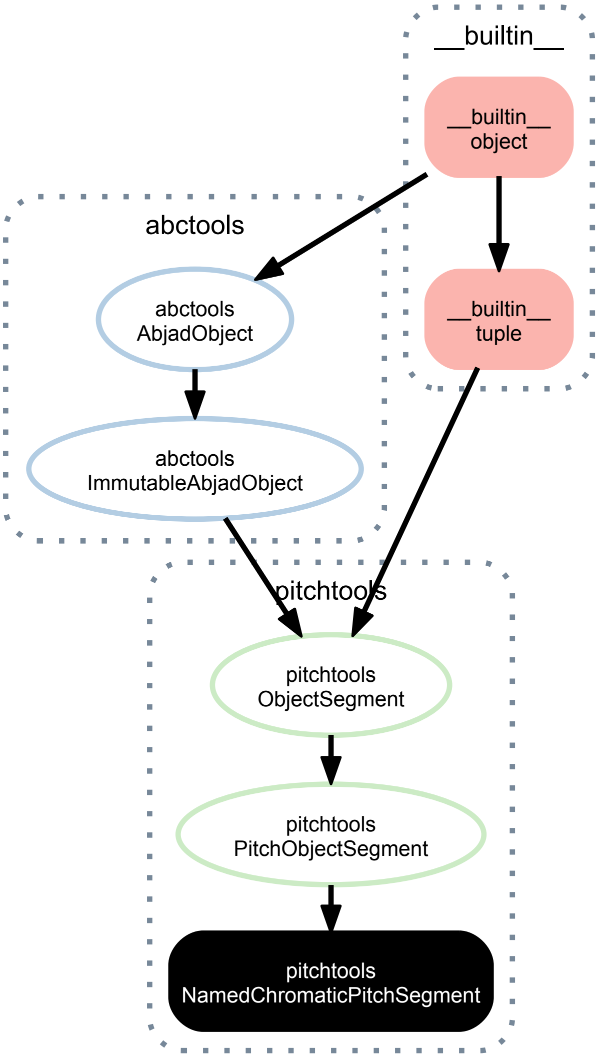
NamedChromaticPitchClassSet.**__rxor__**()
    x.__rxor__(y) <==> y^x

NamedChromaticPitchClassSet.**__str__**()

NamedChromaticPitchClassSet.**__sub__**()
    x.__sub__(y) <==> x-y

NamedChromaticPitchClassSet.**__xor__**()
    x.__xor__(y) <==> x^y

21.2.37 pitchtools.NamedChromaticPitchSegment



class `pitchtools.`**`NamedChromaticPitchSegment`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of a named chromatic pitch segment:

```
>>> pitchtools.NamedChromaticPitchSegment(['bf', 'bqf', "fs'", "g'", 'bqf', "g'"])
NamedChromaticPitchSegment("bf bqf fs' g' bqf g'")
```

Named chromtic pitch segments are immutable.

Read-only properties

`NamedChromaticPitchSegment.`**`chromatic_pitch_numbers`**

`NamedChromaticPitchSegment.`**`harmonic_chromatic_interval_class_segment`**

`NamedChromaticPitchSegment.`**`harmonic_chromatic_interval_segment`**

`NamedChromaticPitchSegment.`**`harmonic_diatonic_interval_class_segment`**

`NamedChromaticPitchSegment.`**`harmonic_diatonic_interval_segment`**

`NamedChromaticPitchSegment.`**`inflection_point_count`**

`NamedChromaticPitchSegment.`**`inversion_equivalent_chromatic_interval_class_segment`**

`NamedChromaticPitchSegment.`**`inversion_equivalent_chromatic_interval_class_set`**

NamedChromaticPitchSegment.**inversion_equivalent_chromatic_interval_class_vector**

NamedChromaticPitchSegment.**local_maxima**

NamedChromaticPitchSegment.**local_minima**

NamedChromaticPitchSegment.**melodic_chromatic_interval_class_segment**

NamedChromaticPitchSegment.**melodic_chromatic_interval_segment**

NamedChromaticPitchSegment.**melodic_diatonic_interval_class_segment**

NamedChromaticPitchSegment.**melodic_diatonic_interval_segment**

NamedChromaticPitchSegment.**named_chromatic_pitch_class_vector**

NamedChromaticPitchSegment.**named_chromatic_pitch_set**

NamedChromaticPitchSegment.**named_chromatic_pitch_vector**

NamedChromaticPitchSegment.**named_chromatic_pitches**

NamedChromaticPitchSegment.**numbered_chromatic_pitch_class_segment**

NamedChromaticPitchSegment.**numbered_chromatic_pitch_class_set**

NamedChromaticPitchSegment.**storage_format**
Storage format of Abjad object.

Return string.

Methods

NamedChromaticPitchSegment.**count** (*value*) → integer – return number of occurrences of value

NamedChromaticPitchSegment.**index** (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

NamedChromaticPitchSegment.**transpose** (*melodic_interval*)
Transpose pitches in pitch segment by melodic interval and emit new pitch segment.

Special methods

NamedChromaticPitchSegment.**__add__** (*arg*)

NamedChromaticPitchSegment.**__contains__** ()
x.__contains__(y) <==> y in x

NamedChromaticPitchSegment.**__eq__** ()
x.__eq__(y) <==> x==y

NamedChromaticPitchSegment.**__ge__** ()
x.__ge__(y) <==> x>=y

NamedChromaticPitchSegment.**__getitem__** ()
x.__getitem__(y) <==> x[y]

NamedChromaticPitchSegment.**__getslice__** (*start*, *stop*)

NamedChromaticPitchSegment.**__gt__** ()
x.__gt__(y) <==> x>y

NamedChromaticPitchSegment.**__hash__** () <==> *hash(x)*

NamedChromaticPitchSegment.**__iter__** () <==> *iter(x)*

NamedChromaticPitchSegment.**__le__** ()
x.__le__(y) <==> x<=y

NamedChromaticPitchSegment.**__len__**() <==> *len(x)*

NamedChromaticPitchSegment.**__lt__**()
    x.__lt__(y) <==> x<y

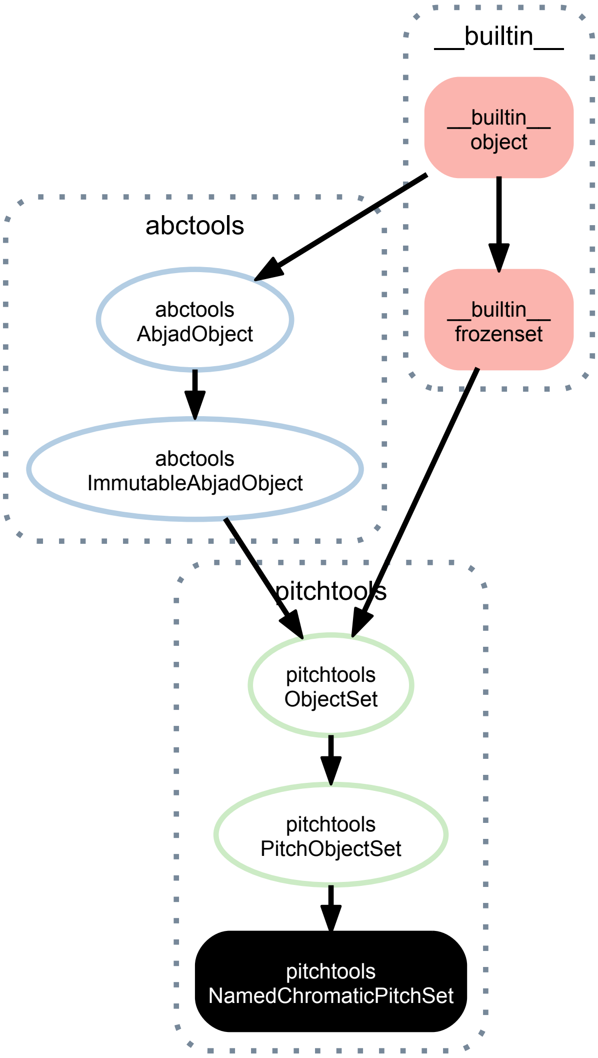
NamedChromaticPitchSegment.**__mul__**(*n*)

NamedChromaticPitchSegment.**__ne__**()
    x.__ne__(y) <==> x!=y

NamedChromaticPitchSegment.**__repr__**()

NamedChromaticPitchSegment.**__rmul__**(*n*)

21.2.38 pitchtools.NamedChromaticPitchSet



class pitchtools.**NamedChromaticPitchSet**(**args*, ***kwargs*)
    New in version 2.0. Abjad model of a named chromatic pitch set:

```
>>> pitchtools.NamedChromaticPitchSet(['bf', 'bqf', "fs'", "g'", 'bqf', "g'"])
NamedChromaticPitchSet(['bf', 'bqf', "fs'", "g'"])
```

Named chromatic pitch sets are immutable.

Read-only properties

NamedChromaticPitchSet.**chromatic_pitch_numbers**

NamedChromaticPitchSet.**duplicate_pitch_classes**

NamedChromaticPitchSet.**is_pitch_class_unique**

NamedChromaticPitchSet.**named_chromatic_pitches**

NamedChromaticPitchSet.**numbered_chromatic_pitch_class_set**

NamedChromaticPitchSet.**numbered_chromatic_pitch_classes**

NamedChromaticPitchSet.**storage_format**
Storage format of Abjad object.

Return string.

Methods

NamedChromaticPitchSet.**copy**()
Return a shallow copy of a set.

NamedChromaticPitchSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

NamedChromaticPitchSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

NamedChromaticPitchSet.**isdisjoint**()
Return True if two sets have a null intersection.

NamedChromaticPitchSet.**issubset**()
Report whether another set contains this set.

NamedChromaticPitchSet.**issuperset**()
Report whether this set contains another set.

NamedChromaticPitchSet.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

NamedChromaticPitchSet.**transpose**(*n*)
Transpose all pcs in self by n.

NamedChromaticPitchSet.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

NamedChromaticPitchSet.**__and__**()
x.__and__(y) <==> x&y

NamedChromaticPitchSet.**__cmp__**(*y*) <==> *cmp(x, y)*

NamedChromaticPitchSet.**__contains__**()
x.__contains__(y) <==> y in x.

NamedChromaticPitchSet.**__eq__**(*arg*)

`NamedChromaticPitchSet.__ge__()`
x.__ge__(y) <==> x>=y

`NamedChromaticPitchSet.__gt__()`
x.__gt__(y) <==> x>y

`NamedChromaticPitchSet.__hash__()` <==> *hash(x)*

`NamedChromaticPitchSet.__iter__()` <==> *iter(x)*

`NamedChromaticPitchSet.__le__()`
x.__le__(y) <==> x<=y

`NamedChromaticPitchSet.__len__()` <==> *len(x)*

`NamedChromaticPitchSet.__lt__()`
x.__lt__(y) <==> x<y

`NamedChromaticPitchSet.__ne__(arg)`

`NamedChromaticPitchSet.__or__()`
x.__or__(y) <==> x|y

`NamedChromaticPitchSet.__rand__()`
x.__rand__(y) <==> y&x

`NamedChromaticPitchSet.__repr__()`

`NamedChromaticPitchSet.__ror__()`
x.__ror__(y) <==> y|x

`NamedChromaticPitchSet.__rsub__()`
x.__rsub__(y) <==> y-x

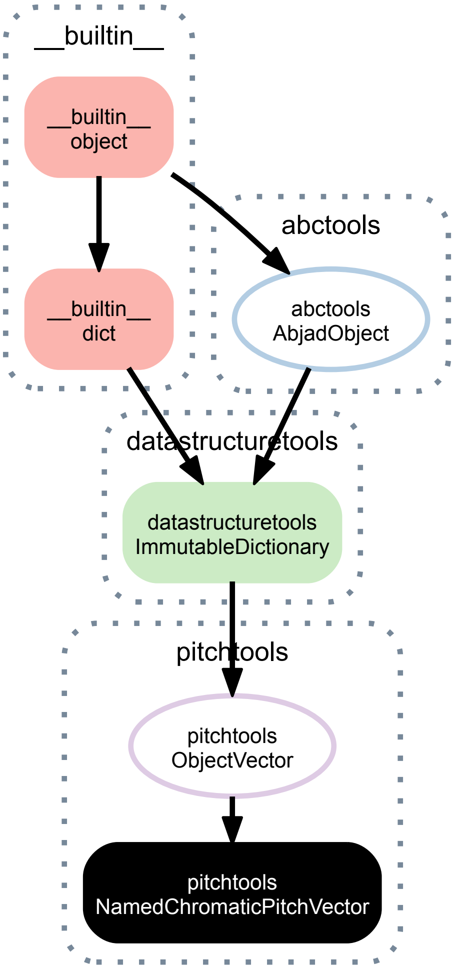
`NamedChromaticPitchSet.__rxor__()`
x.__rxor__(y) <==> y^x

`NamedChromaticPitchSet.__str__()`

`NamedChromaticPitchSet.__sub__()`
x.__sub__(y) <==> x-y

`NamedChromaticPitchSet.__xor__()`
x.__xor__(y) <==> x^y

21.2.39 pitchtools.NamedChromaticPitchVector



class pitchtools.**NamedChromaticPitchVector** (*pitches*)

New in version 2.0. Abjad model of named chromatic pitch vector:

```
>>> named_chromatic_pitch_vector = pitchtools.NamedChromaticPitchVector(
...     ["c''", "c''", "cs''", "cs''", "cs''"])
```

```
>>> named_chromatic_pitch_vector
NamedChromaticPitchVector(c'': 2, cs'': 3)
```

```
>>> print named_chromatic_pitch_vector
NamedChromaticPitchVector(c'': 2, cs'': 3)
```

Named chromatic pitch vectors are immutable.

Read-only properties

NamedChromaticPitchVector.**chromatic_pitch_numbers**

NamedChromaticPitchVector.**named_chromatic_pitches**

NamedChromaticPitchVector.**storage_format**

Storage format of Abjad object.

Return string.

Methods

`NamedChromaticPitchVector.clear()` → None. Remove all items from D.

`NamedChromaticPitchVector.copy()` → a shallow copy of D

`NamedChromaticPitchVector.get(k[, d])` → D[k] if k in D, else d. d defaults to None.

`NamedChromaticPitchVector.has_key(k)` → True if D has a key k, else False

`NamedChromaticPitchVector.items()` → list of D's (key, value) pairs, as 2-tuples

`NamedChromaticPitchVector.iteritems()` → an iterator over the (key, value) items of D

`NamedChromaticPitchVector.iterkeys()` → an iterator over the keys of D

`NamedChromaticPitchVector.itervalues()` → an iterator over the values of D

`NamedChromaticPitchVector.keys()` → list of D's keys

`NamedChromaticPitchVector.pop(k[, d])` → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

`NamedChromaticPitchVector.popitem()` → (k, v), remove and return some (key, value) pair as a

2-tuple; but raise KeyError if D is empty.

`NamedChromaticPitchVector.setdefault(k[, d])` → D.get(k,d), also set D[k]=d if k not in D

`NamedChromaticPitchVector.update([E], **F)` → None. Update D from dict/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`NamedChromaticPitchVector.values()` → list of D's values

`NamedChromaticPitchVector.viewitems()` → a set-like object providing a view on D's items

`NamedChromaticPitchVector.viewkeys()` → a set-like object providing a view on D's keys

`NamedChromaticPitchVector.viewvalues()` → an object providing a view on D's values

Special methods

`NamedChromaticPitchVector.__cmp__(y)` <==> *cmp(x, y)*

`NamedChromaticPitchVector.__contains__(k)` → True if D has a key k, else False

`NamedChromaticPitchVector.__delitem__(*args)`

`NamedChromaticPitchVector.__eq__()`

x.__eq__(y) <==> x==y

`NamedChromaticPitchVector.__ge__()`

x.__ge__(y) <==> x>=y

`NamedChromaticPitchVector.__getitem__()`

x.__getitem__(y) <==> x[y]

`NamedChromaticPitchVector.__gt__()`

x.__gt__(y) <==> x>y

`NamedChromaticPitchVector.__iter__()` <==> *iter(x)*

`NamedChromaticPitchVector.__le__()`

x.__le__(y) <==> x<=y

`NamedChromaticPitchVector.__len__()` <==> *len(x)*

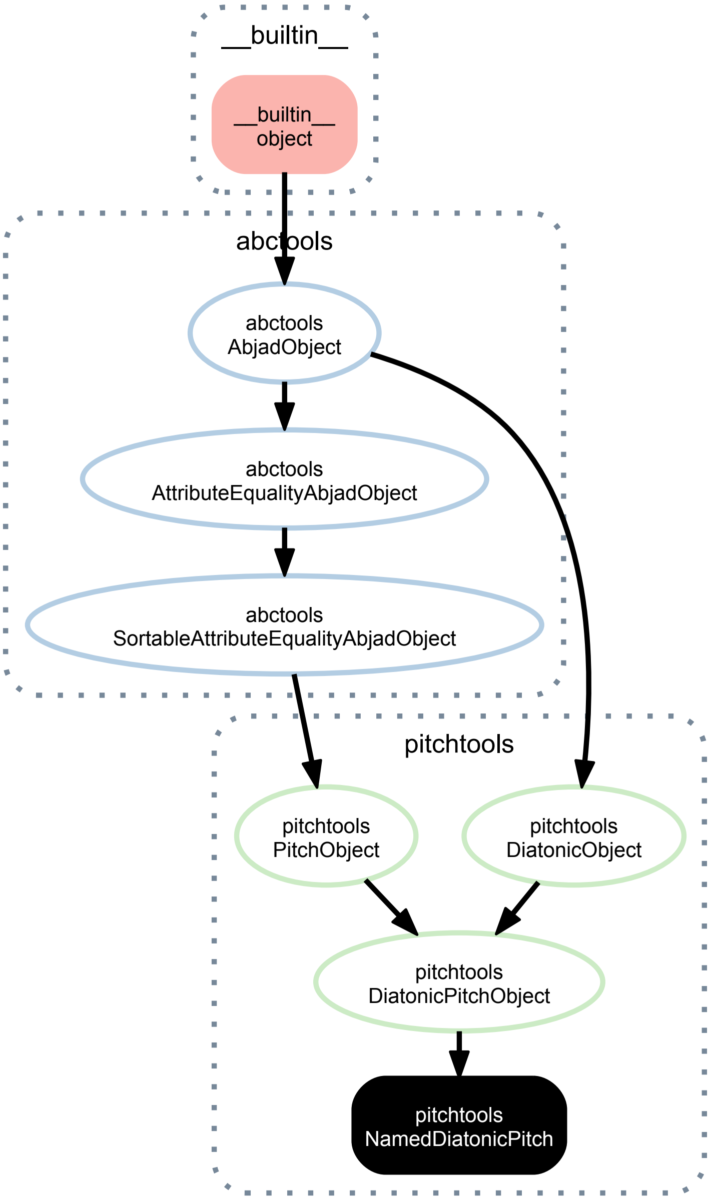
NamedChromaticPitchVector.**__lt__**()
x.__lt__(y) <==> x<y

NamedChromaticPitchVector.**__ne__**()
x.__ne__(y) <==> x!=y

NamedChromaticPitchVector.**__repr__**()

NamedChromaticPitchVector.**__setitem__**(**args*)

21.2.40 pitchtools.NamedDiatonicPitch



class pitchtools.**NamedDiatonicPitch**(*arg*)
New in version 2.0. Abjad model of a named diatonic pitch:

```
>>> named_diatonic_pitch = pitchtools.NamedDiatonicPitch("c''")
```

```
>>> named_diatonic_pitch
NamedDiatonicPitch("c''")
```

```
>>> print named_diatonic_pitch
c''
```

Named diatonic pitches are immutable.

Read-only properties

NamedDiatonicPitch.**chromatic_pitch_class_name**
Read-only chromatic pitch-class name:

```
>>> pitchtools.NamedDiatonicPitch("c''").chromatic_pitch_class_name
'c'
```

Return string.

NamedDiatonicPitch.**chromatic_pitch_class_number**
Read-only chromatic pitch-class number:

```
>>> pitchtools.NamedDiatonicPitch("c''").chromatic_pitch_class_number
0
```

Return integer.

NamedDiatonicPitch.**chromatic_pitch_name**
Read-only chromatic pitch name:

```
>>> pitchtools.NamedDiatonicPitch("c''").chromatic_pitch_name
"c''"
```

Return string.

NamedDiatonicPitch.**chromatic_pitch_number**
Read-only chromatic pitch number:

```
>>> pitchtools.NamedDiatonicPitch("c''").chromatic_pitch_number
12
```

Return integer.

NamedDiatonicPitch.**diatonic_pitch_class_name**
Read-only diatonic pitch-class name:

```
>>> pitchtools.NamedDiatonicPitch("c''").diatonic_pitch_class_name
'c'
```

Return string.

NamedDiatonicPitch.**diatonic_pitch_class_number**
Read-only diatonic pitch-class number:

```
>>> pitchtools.NamedDiatonicPitch("c''").diatonic_pitch_class_number
0
```

Return integer.

NamedDiatonicPitch.**diatonic_pitch_name**
Read-only diatonic pitch name:

```
>>> pitchtools.NamedDiatonicPitch("c''").diatonic_pitch_name
"c''"
```

Return string.

NamedDiatonicPitch.**diatonic_pitch_number**
Read-only diatonic pitch number:

```
>>> pitchtools.NamedDiatonicPitch("c''").diatonic_pitch_number
7
```

Return integer.

NamedDiatonicPitch.**lilypond_format**

Read-only LilyPond input format of named diatonic pitch:

```
>>> pitchtools.NamedDiatonicPitch("c''").lilypond_format
"c''"
```

Return string.

NamedDiatonicPitch.**named_chromatic_pitch**

Read-only named chromatic pitch:

```
>>> pitchtools.NamedDiatonicPitch("c''").named_chromatic_pitch
NamedChromaticPitch("c''")
```

Return named chromatic pitch.

NamedDiatonicPitch.**named_chromatic_pitch_class**

Read-only named chromatic pitch-class:

```
>>> pitchtools.NamedDiatonicPitch("c''").named_chromatic_pitch_class
NamedChromaticPitchClass('c')
```

Return named chromatic pitch-class.

NamedDiatonicPitch.**named_diatonic_pitch_class**

Read-only named diatonic pitch-class:

```
>>> pitchtools.NamedDiatonicPitch("c''").named_diatonic_pitch_class
NamedDiatonicPitchClass('c')
```

Return named diatonic pitch-class.

NamedDiatonicPitch.**numbered_chromatic_pitch**

Read-only numbered chromatic pitch:

```
>>> pitchtools.NamedDiatonicPitch("c''").numbered_chromatic_pitch
NumberedChromaticPitch(12)
```

Return numbered chromatic pitch.

NamedDiatonicPitch.**numbered_chromatic_pitch_class**

Read-only numbered chromatic pitch-class:

```
>>> pitchtools.NamedDiatonicPitch("c''").numbered_chromatic_pitch_class
NumberedChromaticPitchClass(0)
```

Return numbered chromatic pitch-class.

NamedDiatonicPitch.**numbered_diatonic_pitch**

Read-only numbered diatonic pitch:

```
>>> pitchtools.NamedDiatonicPitch("c''").numbered_diatonic_pitch
NumberedDiatonicPitch(7)
```

Return numbered diatonic pitch.

NamedDiatonicPitch.**numbered_diatonic_pitch_class**

Read-only numbered diatonic pitch-class:

```
>>> pitchtools.NamedDiatonicPitch("c''").numbered_diatonic_pitch_class
NumberedDiatonicPitchClass(0)
```

Return numbered diatonic pitch-class.

NamedDiatonicPitch.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

NamedDiatonicPitch.**__abs__**()

NamedDiatonicPitch.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitch.**__float__**()

NamedDiatonicPitch.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitch.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitch.**__hash__**()

NamedDiatonicPitch.**__int__**()

NamedDiatonicPitch.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitch.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

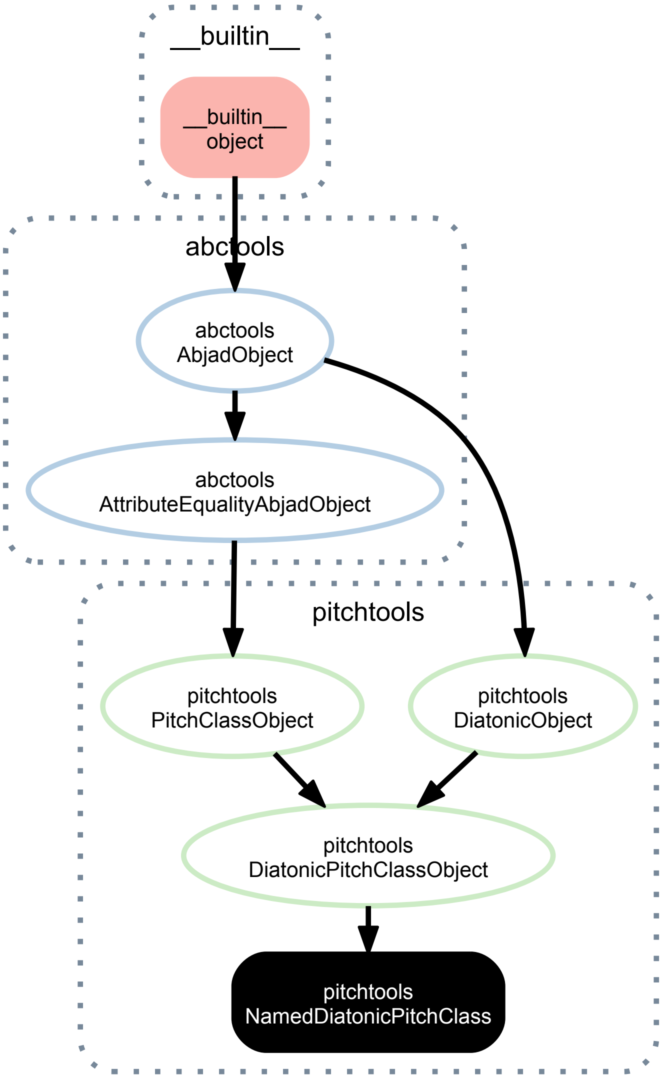
Return boolean.

NamedDiatonicPitch.**__ne__**(*arg*)

NamedDiatonicPitch.**__repr__**()

NamedDiatonicPitch.**__str__**()

21.2.41 pitchtools.NamedDiatonicPitchClass



class pitchtools.**NamedDiatonicPitchClass**(*arg*)

New in version 2.0. Abjad model of a named diatonic pitch-class:

```
>>> pitchtools.NamedDiatonicPitchClass('c')
NamedDiatonicPitchClass('c')
```

Named diatonic pitch-classes are immutable.

Read-only properties

NamedDiatonicPitchClass.**numbered_diatonic_pitch_class**

Read-only numbered diatonic pitch-class from named diatonic pitch-class:

```
>>> named_diatonic_pitch_class = pitchtools.NamedDiatonicPitchClass('c')
>>> named_diatonic_pitch_class.numbered_diatonic_pitch_class
NumberedDiatonicPitchClass(0)
```

Return numbered diatonic pitch-class.

NamedDiatonicPitchClass.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

NamedDiatonicPitchClass.**__abs__**()

NamedDiatonicPitchClass.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitchClass.**__float__**()

NamedDiatonicPitchClass.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NamedDiatonicPitchClass.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

NamedDiatonicPitchClass.**__hash__**()

NamedDiatonicPitchClass.**__int__**()

NamedDiatonicPitchClass.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NamedDiatonicPitchClass.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

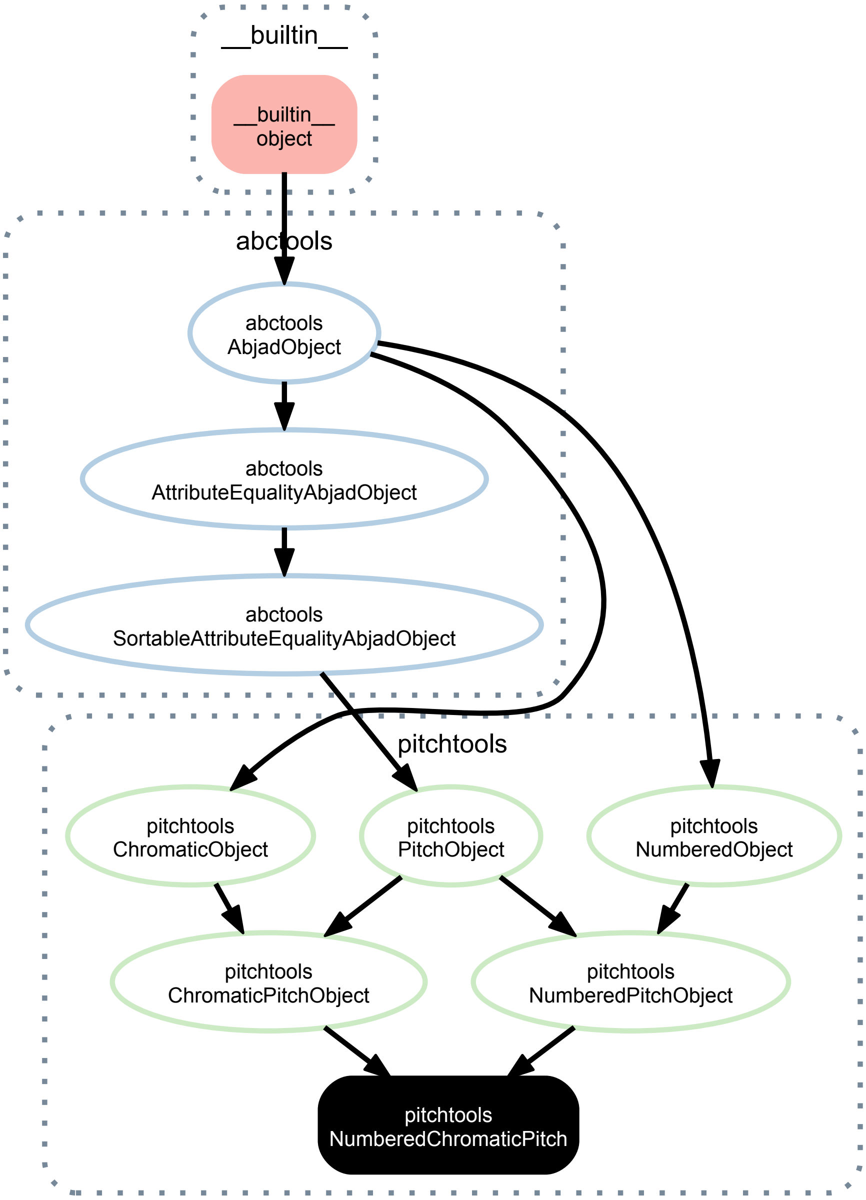
NamedDiatonicPitchClass.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NamedDiatonicPitchClass.**__repr__**()

NamedDiatonicPitchClass.**__str__**()

21.2.42 pitchtools.NumberedChromaticPitch



class pitchtools.**NumberedChromaticPitch**(*arg*)

New in version 2.0. Abjad model of a numbered chromatic pitch:

```
>>> pitchtools.NumberedChromaticPitch(13)
NumberedChromaticPitch(13)
```

Numbered chromatic pitches are immutable.

Read-only properties

NumberedChromaticPitch.**chromatic_pitch_number**

Read-only chromatic pitch-class number:

```
>>> pitchtools.NumberedChromaticPitch(13).chromatic_pitch_number
13
```

Return integer or float.

NumberedChromaticPitch.**diatonic_pitch_class_number**

Read-only diatonic pitch-class number:

```
>>> pitchtools.NumberedChromaticPitch(13).diatonic_pitch_class_number
0
```

Return integer.

NumberedChromaticPitch.**diatonic_pitch_number**

Read-only diatonic pitch-class number:

```
>>> pitchtools.NumberedChromaticPitch(13).diatonic_pitch_number
7
```

Return integer.

NumberedChromaticPitch.**storage_format**

Storage format of Abjad object.

Return string.

Methods

NumberedChromaticPitch.**apply_accidental**(*accidental=None*)

Apply *accidental*:

```
>>> pitchtools.NumberedChromaticPitch(13).apply_accidental('flat')
NumberedChromaticPitch(12)
```

Return numbered chromatic pitch.

NumberedChromaticPitch.**transpose**(*n=0*)

Tranpose by *n* semitones:

```
>>> pitchtools.NumberedChromaticPitch(13).transpose(1)
NumberedChromaticPitch(14)
```

Return numbered chromatic pitch.

Special methods

NumberedChromaticPitch.**__abs__**()

NumberedChromaticPitch.**__add__**(*arg*)

NumberedChromaticPitch.**__eq__**(*arg*)

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedChromaticPitch.**__float__**()

NumberedChromaticPitch.**__ge__**(*arg*)

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedChromaticPitch.**__gt__**(*arg*)

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedChromaticPitch.**__hash__**()

NumberedChromaticPitch.**__int__**()

NumberedChromaticPitch.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedChromaticPitch.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedChromaticPitch.**__ne__**(*arg*)

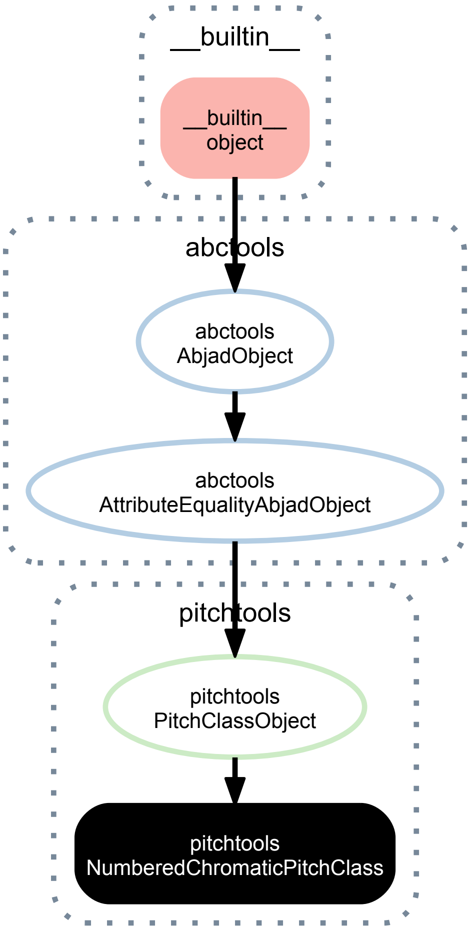
NumberedChromaticPitch.**__neg__**()

NumberedChromaticPitch.**__repr__**()

NumberedChromaticPitch.**__str__**()

NumberedChromaticPitch.**__sub__**(*arg*)

21.2.43 pitchtools.NumberedChromaticPitchClass



class pitchtools.**NumberedChromaticPitchClass**(*arg*)
New in version 2.0. Abjad model of a numbered chromatic pitch-class:

```
>>> pitchtools.NumberedChromaticPitchClass(13)
NumberedChromaticPitchClass(1)
```

Numbered chromatic pitch-classes are immutable.

Read-only properties

`NumberedChromaticPitchClass.`**storage_format**
Storage format of Abjad object.

Return string.

Methods

`NumberedChromaticPitchClass.`**apply_accidental**(*accidental=None*)
Emit new numbered chromatic pitch-class as sum of self and accidental.

`NumberedChromaticPitchClass.`**invert**()
Invert pitch-class.

`NumberedChromaticPitchClass.`**multiply**(*n*)
Multiply pitch-class by n.

`NumberedChromaticPitchClass.`**transpose**(*n*)
Transpose pitch-class by n.

Special methods

`NumberedChromaticPitchClass.`**__abs__**()

`NumberedChromaticPitchClass.`**__add__**(*arg*)
Addition defined against melodic chromatic intervals only.

`NumberedChromaticPitchClass.`**__copy__**(**args*)

`NumberedChromaticPitchClass.`**__deepcopy__**(**args*)

`NumberedChromaticPitchClass.`**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`NumberedChromaticPitchClass.`**__float__**()

`NumberedChromaticPitchClass.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`NumberedChromaticPitchClass.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`NumberedChromaticPitchClass.`**__hash__**()

`NumberedChromaticPitchClass.`**__int__**()

`NumberedChromaticPitchClass.`**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`NumberedChromaticPitchClass.`**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`NumberedChromaticPitchClass.`**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

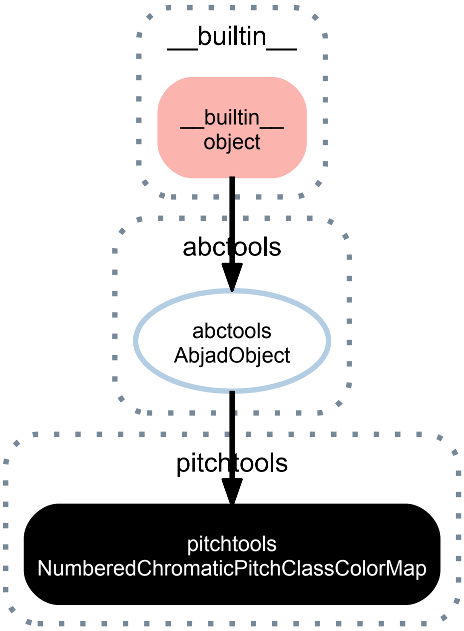
NumberedChromaticPitchClass.**__neg__**()

NumberedChromaticPitchClass.**__repr__**()

NumberedChromaticPitchClass.**__str__**()

NumberedChromaticPitchClass.**__sub__**(*arg*)
Subtraction defined against both melodic chromatic intervals and against other pitch-classes.

21.2.44 pitchtools.NumberedChromaticPitchClassColorMap



class pitchtools.**NumberedChromaticPitchClassColorMap**(*pitch_iterables*, *colors*)
New in version 2.0. Abjad model of a numbered chromatic pitch-class color map:

```
>>> chromatic_pitch_class_numbers = [[-8, 2, 10, 21], [0, 11, 32, 41], [15, 25, 42, 43]]
>>> colors = ['red', 'green', 'blue']
>>> mapping = pitchtools.NumberedChromaticPitchClassColorMap(
... chromatic_pitch_class_numbers, colors)
```

Numbered chromatic pitch-class color maps are immutable.

Read-only properties

NumberedChromaticPitchClassColorMap.**colors**

NumberedChromaticPitchClassColorMap.**pairs**

NumberedChromaticPitchClassColorMap.**pitch_iterables**

NumberedChromaticPitchClassColorMap.**storage_format**
Storage format of Abjad object.

Return string.

NumberedChromaticPitchClassColorMap.**twelve_tone_complete**

NumberedChromaticPitchClassColorMap.**twenty_four_tone_complete**

Methods

NumberedChromaticPitchClassColorMap.**get**(*key*, *alternative=None*)

Special methods

`NumberedChromaticPitchClassColorMap.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`NumberedChromaticPitchClassColorMap.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NumberedChromaticPitchClassColorMap.__getitem__(pc)`

`NumberedChromaticPitchClassColorMap.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`NumberedChromaticPitchClassColorMap.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NumberedChromaticPitchClassColorMap.__lt__(expr)`
Abjad objects by default do not implement this method.

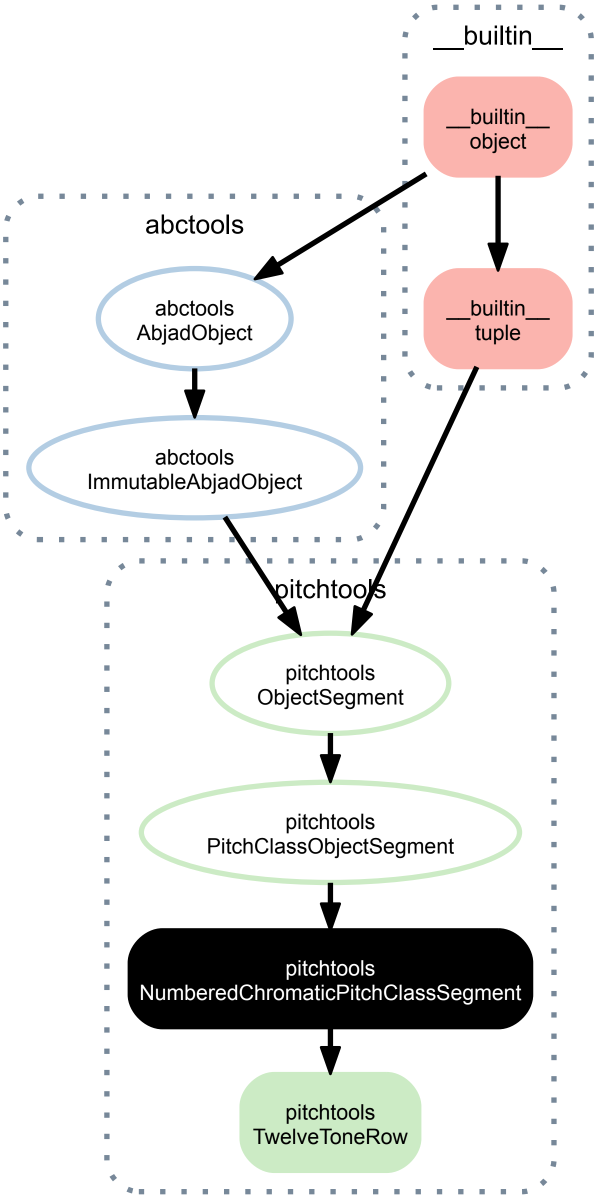
Raise exception.

`NumberedChromaticPitchClassColorMap.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`NumberedChromaticPitchClassColorMap.__repr__()`

21.2.45 pitchtools.NumberedChromaticPitchClassSegment



class `pitchtools.`**`NumberedChromaticPitchClassSegment`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of a numbered chromatic pitch-class segment:

```
>>> pitchtools.NumberedChromaticPitchClassSegment([-2, -1.5, 6, 7, -1.5, 7])
NumberedChromaticPitchClassSegment([10, 10.5, 6, 7, 10.5, 7])
```

Numbered chromatic pitch-class segments are immutable.

Read-only properties

`NumberedChromaticPitchClassSegment.`**`inversion_equivalent_chromatic_interval_class_segment`**

Read-only inversion-equivalent chromatic interval-class segment:

```
>>> segment = pitchtools.NumberedChromaticPitchClassSegment([10, 10.5, 6, 7, 10.5, 7])
```

```
>>> segment.inversion_equivalent_chromatic_interval_class_segment
InversionEquivalentChromaticIntervalClassSegment(0.5, 4.5, 1, 3.5, 3.5)
```

Return inversion-equivalent chromatic interval-class segment.

NumberedChromaticPitchClassSegment.**numbered_chromatic_pitch_class_set**
Read-only numbered chromatic pitch-class set from numbered chromatic pitch-class segment:

```
>>> segment.numbered_chromatic_pitch_class_set
NumberedChromaticPitchClassSet([6, 7, 10, 10.5])
```

Return numbered chromatic pitch-class set.

NumberedChromaticPitchClassSegment.**storage_format**
Storage format of Abjad object.

Return string.

Methods

NumberedChromaticPitchClassSegment.**alpha**()
Morris alpha transform of numbered chromatic pitch-class segment:

```
>>> segment.alpha()
NumberedChromaticPitchClassSegment([11, 11.5, 7, 6, 11.5, 6])
```

Return numbered chromatic pitch-class segment.

NumberedChromaticPitchClassSegment.**count**(*value*) → integer – return number of occurrences of value

NumberedChromaticPitchClassSegment.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

NumberedChromaticPitchClassSegment.**invert**()
Invert numbered chromatic pitch-class segment:

```
>>> segment.invert()
NumberedChromaticPitchClassSegment([2, 1.5, 6, 5, 1.5, 5])
```

Return numbered chromatic pitch-class segment.

NumberedChromaticPitchClassSegment.**multiply**(*n*)
Multiply numbered chromatic pitch-class segment by *n*:

```
>>> segment.multiply(5)
NumberedChromaticPitchClassSegment([2, 4.5, 6, 11, 4.5, 11])
```

Return numbered chromatic pitch-class segment.

NumberedChromaticPitchClassSegment.**retrograde**()
Retrograde of numbered chromatic pitch-class segment:

```
>>> segment.retrograde()
NumberedChromaticPitchClassSegment([7, 10.5, 7, 6, 10.5, 10])
```

Return numbered chromatic pitch-class segment.

NumberedChromaticPitchClassSegment.**rotate**(*n*)
Rotate numbered chromatic pitch-class segment:

```
>>> segment.rotate(1)
NumberedChromaticPitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

Return numbered chromatic pitch-class segment.

NumberedChromaticPitchClassSegment.**transpose**(*n*)
Transpose numbered chromatic pitch-class segment:

```
>>> segment.transpose(10)
NumberedChromaticPitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Return numbered chromatic pitch-class segment.

Special methods

NumberedChromaticPitchClassSegment.**__add__**(*arg*)

NumberedChromaticPitchClassSegment.**__contains__**()
x.__contains__(y) <==> y in x

NumberedChromaticPitchClassSegment.**__eq__**()
x.__eq__(y) <==> x==y

NumberedChromaticPitchClassSegment.**__ge__**()
x.__ge__(y) <==> x>=y

NumberedChromaticPitchClassSegment.**__getitem__**()
x.__getitem__(y) <==> x[y]

NumberedChromaticPitchClassSegment.**__getslice__**(*start*, *stop*)

NumberedChromaticPitchClassSegment.**__gt__**()
x.__gt__(y) <==> x>y

NumberedChromaticPitchClassSegment.**__hash__**() <==> *hash(x)*

NumberedChromaticPitchClassSegment.**__iter__**() <==> *iter(x)*

NumberedChromaticPitchClassSegment.**__le__**()
x.__le__(y) <==> x<=y

NumberedChromaticPitchClassSegment.**__len__**() <==> *len(x)*

NumberedChromaticPitchClassSegment.**__lt__**()
x.__lt__(y) <==> x<y

NumberedChromaticPitchClassSegment.**__mul__**(*n*)

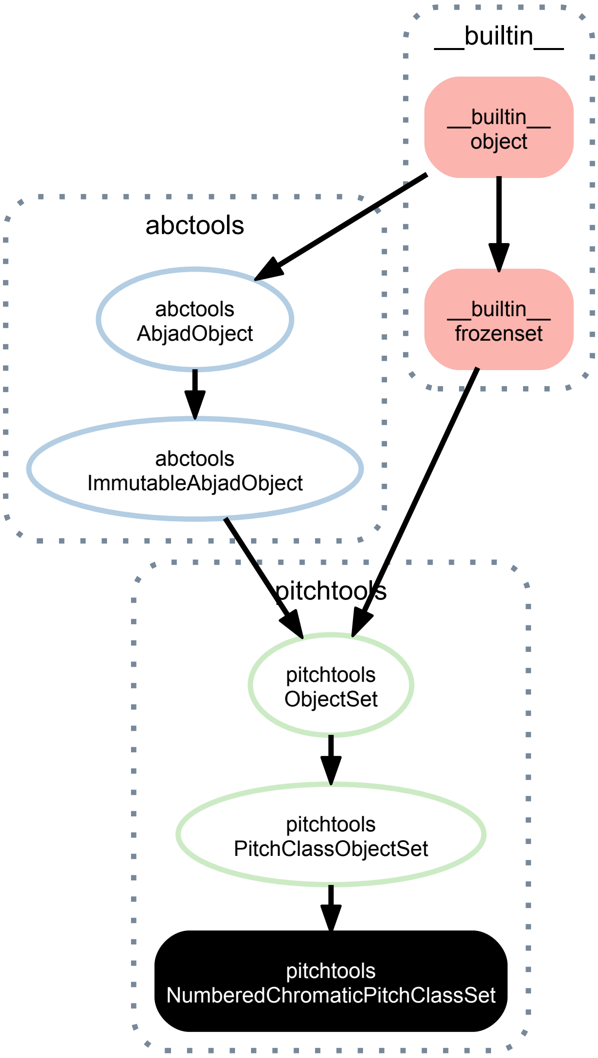
NumberedChromaticPitchClassSegment.**__ne__**()
x.__ne__(y) <==> x!=y

NumberedChromaticPitchClassSegment.**__repr__**()

NumberedChromaticPitchClassSegment.**__rmul__**(*n*)

NumberedChromaticPitchClassSegment.**__str__**()

21.2.46 pitchtools.NumberedChromaticPitchClassSet



class `pitchtools.NumberedChromaticPitchClassSet` (**args*, ***kwargs*)
New in version 2.0. Abjad model of a numbered chromatic pitch-class set:

```
>>> ncpcs = pitchtools.NumberedChromaticPitchClassSet([-2, -1.5, 6, 7, -1.5, 7])
```

```
>>> ncpcs
NumberedChromaticPitchClassSet([6, 7, 10, 10.5])
```

```
>>> print ncpcs
{6, 7, 10, 10.5}
```

Numbered chromatic pitch-class sets are immutable.

Read-only properties

`NumberedChromaticPitchClassSet.`**`inversion_equivalent_chromatic_interval_class_set`**
Read-only inversion-equivalent chromatic interval-class set:

```
>>> ncpcs.inversion_equivalent_chromatic_interval_class_set
InversionEquivalentChromaticIntervalClassSet(0.5, 1, 3, 3.5, 4, 4.5)
```

Return inversion-equivalent chromatic interval-class set.

`NumberedChromaticPitchClassSet.`**`inversion_equivalent_chromatic_interval_class_vector`**
Read-only inversion-equivalent chromatic interval-class vector:

```
>>> ncpcs.inversion_equivalent_chromatic_interval_class_vector
InversionEquivalentChromaticIntervalClassVector(0 | 1 0 1 1 0 0 1 0 0 1 1 0)
```

Return inversion-equivalent chromatic interval-class vector.

NumberedChromaticPitchClassSet.**numbered_chromatic_pitch_classes**
Read-only numbered chromatic pitch-classes:

```
>>> result = ncpcs.numbered_chromatic_pitch_classes
```

```
>>> for x in result: x
...
NumberedChromaticPitchClass(6)
NumberedChromaticPitchClass(7)
NumberedChromaticPitchClass(10)
NumberedChromaticPitchClass(10.5)
```

Return tuple.

NumberedChromaticPitchClassSet.**prime_form**
To be implemented.

NumberedChromaticPitchClassSet.**storage_format**
Storage format of Abjad object.

Return string.

Methods

NumberedChromaticPitchClassSet.**copy**()
Return a shallow copy of a set.

NumberedChromaticPitchClassSet.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

NumberedChromaticPitchClassSet.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

NumberedChromaticPitchClassSet.**invert**()
Invert numbered chromatic pitch-class set:

```
>>> ncpcs.invert()
NumberedChromaticPitchClassSet([1.5, 2, 5, 6])
```

Return numbered chromatic pitch-class set.

NumberedChromaticPitchClassSet.**is_transposed_subset**(*pcset*)
True when self is transposed subset of *pcset*. False otherwise:

```
>>> pcset_1 = pitchtools.NumberedChromaticPitchClassSet(
... [-2, -1.5, 6, 7, -1.5, 7])
>>> pcset_2 = pitchtools.NumberedChromaticPitchClassSet(
... [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8])
```

```
>>> pcset_1.is_transposed_subset(pcset_2)
True
```

Return boolean.

NumberedChromaticPitchClassSet.**is_transposed_superset**(*pcset*)
True when self is transposed superset of *pcset*. False otherwise:

```
>>> pcset_1 = pitchtools.NumberedChromaticPitchClassSet(
... [-2, -1.5, 6, 7, -1.5, 7])
>>> pcset_2 = pitchtools.NumberedChromaticPitchClassSet(
... [-2, -1.5, 6, 7, -1.5, 7, 7.5, 8])
```

```
>>> pcset_2.is_transposed_superset(pcset_1)
True
```

Return boolean.

`NumberedChromaticPitchClassSet.`**`isdisjoint`**`()`
Return True if two sets have a null intersection.

`NumberedChromaticPitchClassSet.`**`issubset`**`()`
Report whether another set contains this set.

`NumberedChromaticPitchClassSet.`**`issuperset`**`()`
Report whether this set contains another set.

`NumberedChromaticPitchClassSet.`**`multiply`**`(`*n*`)`
Multiply numbered chromatic pitch-class set by *n*:

```
>>> ncpcs.multiply(5)
NumberedChromaticPitchClassSet([2, 4.5, 6, 11])
```

Return numbered chromatic pitch-class set.

`NumberedChromaticPitchClassSet.`**`symmetric_difference`**`()`
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

`NumberedChromaticPitchClassSet.`**`transpose`**`(`*n*`)`
Transpose numbered chromatic pitch-class set by *n*:

```
>>> ncpcs.transpose(5)
NumberedChromaticPitchClassSet([0, 3, 3.5, 11])
```

Return numbered chromatic pitch-class set.

`NumberedChromaticPitchClassSet.`**`union`**`()`
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

`NumberedChromaticPitchClassSet.`**`__and__`**`()`
x.__and__(y) <==> x&y

`NumberedChromaticPitchClassSet.`**`__cmp__`**`(`*y*`)` <==> *cmp(x, y)*

`NumberedChromaticPitchClassSet.`**`__contains__`**`()`
x.__contains__(y) <==> y in x.

`NumberedChromaticPitchClassSet.`**`__eq__`**`(`*arg*`)`

`NumberedChromaticPitchClassSet.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`NumberedChromaticPitchClassSet.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`NumberedChromaticPitchClassSet.`**`__hash__`**`()`

`NumberedChromaticPitchClassSet.`**`__iter__`**`()` <==> *iter(x)*

`NumberedChromaticPitchClassSet.`**`__le__`**`()`
x.__le__(y) <==> x<=y

NumberedChromaticPitchClassSet.**__len__**() <==> *len(x)*

NumberedChromaticPitchClassSet.**__lt__**()
x.__lt__(y) <==> x<y

NumberedChromaticPitchClassSet.**__ne__**(*arg*)

NumberedChromaticPitchClassSet.**__or__**()
x.__or__(y) <==> x|y

NumberedChromaticPitchClassSet.**__rand__**()
x.__rand__(y) <==> y&x

NumberedChromaticPitchClassSet.**__repr__**()

NumberedChromaticPitchClassSet.**__ror__**()
x.__ror__(y) <==> y|x

NumberedChromaticPitchClassSet.**__rsub__**()
x.__rsub__(y) <==> y-x

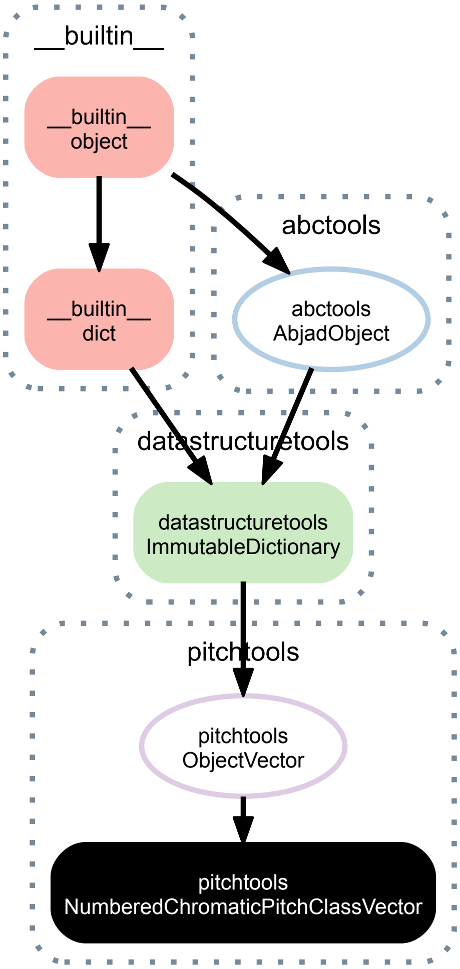
NumberedChromaticPitchClassSet.**__rxor__**()
x.__rxor__(y) <==> y^x

NumberedChromaticPitchClassSet.**__str__**()

NumberedChromaticPitchClassSet.**__sub__**()
x.__sub__(y) <==> x-y

NumberedChromaticPitchClassSet.**__xor__**()
x.__xor__(y) <==> x^y

21.2.47 pitchtools.NumberedChromaticPitchClassVector



class `pitchtools.`**`NumberedChromaticPitchClassVector`**(*pitch_class_tokens*)

New in version 2.0. Abjad model of numbered chromatic pitch-class vector:

```
>>> ncpcv = pitchtools.NumberedChromaticPitchClassVector(
...     [13, 13, 14.5, 14.5, 14.5, 6, 6, 6])
```

```
>>> print ncpcv
0 2 0 0 0 0 | 3 0 0 0 0 0
0 0 3 0 0 0 | 0 0 0 0 0 0
```

Numbered chromatic pitch-class vectors are immutable.

Read-only properties

`NumberedChromaticPitchClassVector.`**`chromatic_pitch_class_numbers`**

Read-only chromatic pitch-class numbers from numbered chromatic pitch-class vector:

```
>>> ncpcv.chromatic_pitch_class_numbers
[1, 2.5, 6]
```

Return list.

`NumberedChromaticPitchClassVector.`**`numbered_chromatic_pitch_classes`**

Read-only numbered chromatic pitch-classes from numbered chromatic pitch-class vector:

```
>>> result = ncpcv.numbered_chromatic_pitch_classes
```

```
>>> for x in result: x
...
NumberedChromaticPitchClass(2.5)
NumberedChromaticPitchClass(1)
NumberedChromaticPitchClass(6)
```

Return list.

`NumberedChromaticPitchClassVector.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`NumberedChromaticPitchClassVector.`**`clear`**`()` → None. Remove all items from D.

`NumberedChromaticPitchClassVector.`**`copy`**`()` → a shallow copy of D

`NumberedChromaticPitchClassVector.`**`get`**`(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

`NumberedChromaticPitchClassVector.`**`has_key`**`(`*k*`)` → True if D has a key k, else False

`NumberedChromaticPitchClassVector.`**`items`**`()` → list of D's (key, value) pairs, as 2-tuples

`NumberedChromaticPitchClassVector.`**`iteritems`**`()` → an iterator over the (key, value) items of D

`NumberedChromaticPitchClassVector.`**`iterkeys`**`()` → an iterator over the keys of D

`NumberedChromaticPitchClassVector.`**`itervalues`**`()` → an iterator over the values of D

`NumberedChromaticPitchClassVector.`**`keys`**`()` → list of D's keys

`NumberedChromaticPitchClassVector.`**`pop`**`(`*k*[, *d*]`)` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`NumberedChromaticPitchClassVector.`**`popitem`**`()` → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`NumberedChromaticPitchClassVector.`**`setdefault`**`(`*k*[, *d*]`)` → D.get(k,d), also set D[k]=d if k not in D

`NumberedChromaticPitchClassVector.`**`update`**`(`[*E*], ***F*`)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`NumberedChromaticPitchClassVector.`**`values`**`()` → list of D's values

`NumberedChromaticPitchClassVector.`**`viewitems`**`()` → a set-like object providing a view on D's items

`NumberedChromaticPitchClassVector.`**`viewkeys`**`()` → a set-like object providing a view on D's keys

`NumberedChromaticPitchClassVector.`**`viewvalues`**`()` → an object providing a view on D's values

Special methods

`NumberedChromaticPitchClassVector.`**`__cmp__`**`(`*y*`)` <==> *cmp(x, y)*

`NumberedChromaticPitchClassVector.`**`__contains__`**`(`*k*`)` → True if D has a key k, else False

`NumberedChromaticPitchClassVector.`**`__delitem__`**`(`**args*`)`

NumberedChromaticPitchClassVector.**__eq__**()
x.__eq__(y) <==> x==y

NumberedChromaticPitchClassVector.**__ge__**()
x.__ge__(y) <==> x>=y

NumberedChromaticPitchClassVector.**__getitem__**()
x.__getitem__(y) <==> x[y]

NumberedChromaticPitchClassVector.**__gt__**()
x.__gt__(y) <==> x>y

NumberedChromaticPitchClassVector.**__iter__**(*) <==> iter(x*)

NumberedChromaticPitchClassVector.**__le__**()
x.__le__(y) <==> x<=y

NumberedChromaticPitchClassVector.**__len__**(*) <==> len(x*)

NumberedChromaticPitchClassVector.**__lt__**()
x.__lt__(y) <==> x<y

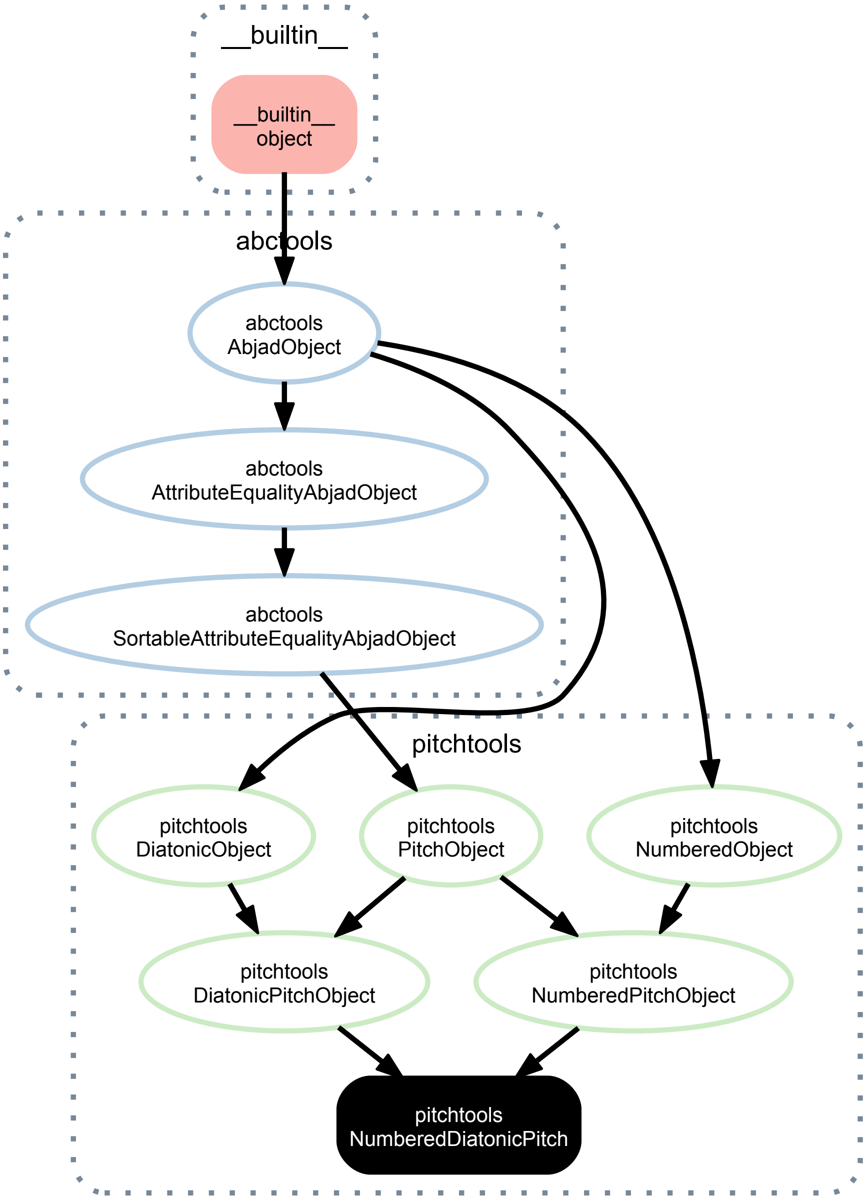
NumberedChromaticPitchClassVector.**__ne__**()
x.__ne__(y) <==> x!=y

NumberedChromaticPitchClassVector.**__repr__**()

NumberedChromaticPitchClassVector.**__setitem__**(**args*)

NumberedChromaticPitchClassVector.**__str__**()

21.2.48 pitchtools.NumberedDiatonicPitch



class `pitchtools.`**`NumberedDiatonicPitch`**(*arg*)

New in version 2.0. Abjad model of a numbered diatonic pitch:

```
>>> pitchtools.NumberedDiatonicPitch(7)
NumberedDiatonicPitch(7)
```

Numbered diatonic pitches are immutable.

Read-only properties

`NumberedDiatonicPitch.`**`chromatic_pitch_number`**

Read-only chromatic pitch number:

```
>>> pitchtools.NumberedDiatonicPitch(7).chromatic_pitch_number
12
```

Return integer.

NumberedDiatonicPitch.**diatonic_pitch_number**
Read-only diatonic pitch number:

```
>>> pitchtools.NumberedDiatonicPitch(7).diatonic_pitch_number
7
```

Return integer.

NumberedDiatonicPitch.**named_diatonic_pitch**
Read-only named diatonic pitch:

```
>>> pitchtools.NumberedDiatonicPitch(7).named_diatonic_pitch
NamedDiatonicPitch("c''")
```

Return named diatonic pitch.

NumberedDiatonicPitch.**named_diatonic_pitch_class**
Read-only named diatonic pitch-class:

```
>>> pitchtools.NumberedDiatonicPitch(7).named_diatonic_pitch_class
NamedDiatonicPitchClass('c')
```

Return named diatonic pitch-class.

NumberedDiatonicPitch.**numbered_diatonic_pitch_class**
Read-only numbered diatonic pitch-class:

```
>>> pitchtools.NumberedDiatonicPitch(7).numbered_diatonic_pitch_class
NumberedDiatonicPitchClass(0)
```

Return numbered diatonic pitch-class.

NumberedDiatonicPitch.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

NumberedDiatonicPitch.**__abs__**()

NumberedDiatonicPitch.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedDiatonicPitch.**__float__**()

NumberedDiatonicPitch.**__ge__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedDiatonicPitch.**__gt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedDiatonicPitch.**__hash__**()

NumberedDiatonicPitch.**__int__**()

NumberedDiatonicPitch.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

NumberedDiatonicPitch.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

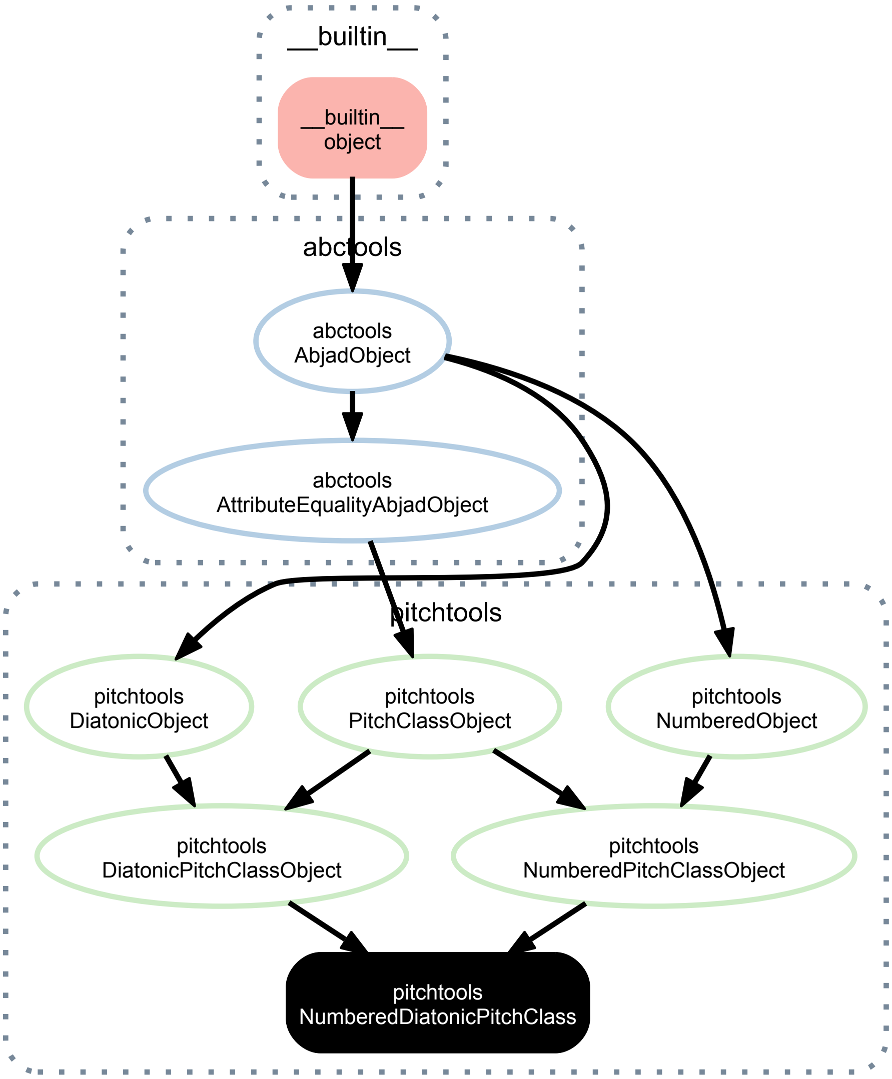
Return boolean.

NumberedDiatonicPitch.**__ne__**(*arg*)

NumberedDiatonicPitch.**__repr__**()

NumberedDiatonicPitch.**__str__**()

21.2.49 pitchtools.NumberedDiatonicPitchClass



class pitchtools.**NumberedDiatonicPitchClass**(*arg*)
New in version 2.0. Abjad model of a numbered diatonic pitch-class:

```
>>> pitchtools.NumberedDiatonicPitchClass(0)
NumberedDiatonicPitchClass(0)
```

Numbered diatonic pitch-classes are immutable.

Read-only properties

NumberedDiatonicPitchClass.**named_diatonic_pitch_class**
Read-only named diatonic pitch-class from numbered diatonic pitch-class:

```
>>> numbered_diatonic_pitch_class = pitchtools.NumberedDiatonicPitchClass(0)
>>> numbered_diatonic_pitch_class.named_diatonic_pitch_class
NamedDiatonicPitchClass('c')
```

Return named diatonic pitch-class.

`NumberedDiatonicPitchClass.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`NumberedDiatonicPitchClass.`**`__abs__`**`()`

`NumberedDiatonicPitchClass.`**`__eq__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`NumberedDiatonicPitchClass.`**`__float__`**`()`

`NumberedDiatonicPitchClass.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`NumberedDiatonicPitchClass.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`NumberedDiatonicPitchClass.`**`__hash__`**`()`

`NumberedDiatonicPitchClass.`**`__int__`**`()`

`NumberedDiatonicPitchClass.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`NumberedDiatonicPitchClass.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

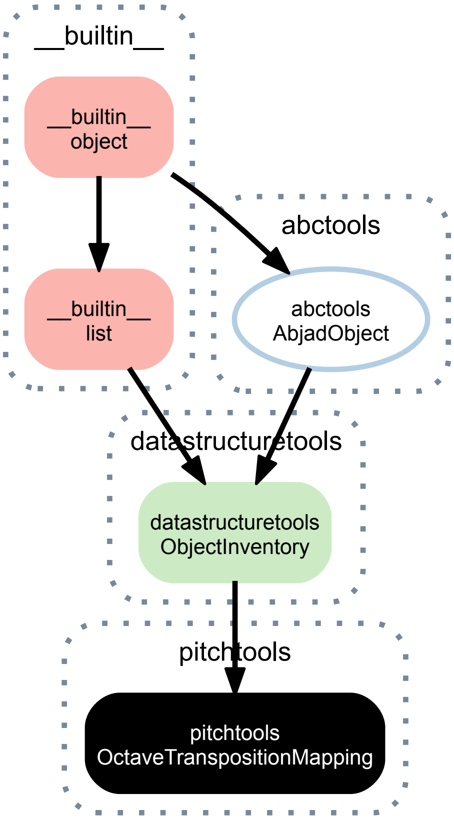
`NumberedDiatonicPitchClass.`**`__ne__`**`(`*arg*`)`
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`NumberedDiatonicPitchClass.`**`__repr__`**`()`

`NumberedDiatonicPitchClass.`**`__str__`**`()`

21.2.50 pitchtools.OctaveTranspositionMapping



class pitchtools.**OctaveTranspositionMapping**(*tokens=None*, *name=None*)

New in version 2.8. Octave transposition mapping:

```
>>> mapping = pitchtools.OctaveTranspositionMapping([('[A0, C4)', 15), ('[C4, C8)', 27)])
```

```
>>> mapping
OctaveTranspositionMapping([('[A0, C4)', 15), ('[C4, C8)', 27)])
```

Octave transposition mappings model `pitchtools.transpose_chromatic_pitch_number_by_octave_tran` input.

Octave transposition mappings implement the list interface and are mutable.

Read-only properties

OctaveTranspositionMapping.**storage_format**

Octave transposition mapping storage format.

```
>>> z(mapping)
pitchtools.OctaveTranspositionMapping([
    pitchtools.OctaveTranspositionMappingComponent(
        pitchtools.PitchRange(
            '[A0, C4)'
            ),
        pitchtools.NumberedChromaticPitch(
            15
            )
        ),
    pitchtools.OctaveTranspositionMappingComponent(
        pitchtools.PitchRange(
            '[C4, C8)'
            ),
        pitchtools.NumberedChromaticPitch(
            27
```

```
                    )
                )
            ])
```

Return string.

Read/write properties

`OctaveTranspositionMapping.name`
Read / write name of inventory.

Methods

`OctaveTranspositionMapping.append(token)`
Change *token* to item and append.

`OctaveTranspositionMapping.count(value)` → integer – return number of occurrences of value

`OctaveTranspositionMapping.extend(tokens)`
Change *tokens* to items and extend.

`OctaveTranspositionMapping.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`OctaveTranspositionMapping.insert()`
L.insert(index, object) – insert object before index

`OctaveTranspositionMapping.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`OctaveTranspositionMapping.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`OctaveTranspositionMapping.reverse()`
L.reverse() – reverse *IN PLACE*

`OctaveTranspositionMapping.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`OctaveTranspositionMapping.__add__()`
x.__add__(y) <==> x+y

`OctaveTranspositionMapping.__call__(pitches)`
Call octave transposition mapping on *pitches*.

```
>>> mapping([-24, -22, -23, -21])
[24, 26, 25, 15]
```

```
>>> mapping([0, 2, 1, 3])
[36, 38, 37, 27]
```

Return list.

`OctaveTranspositionMapping.__contains__(token)`

`OctaveTranspositionMapping.__delitem__()`
x.__delitem__(y) <==> del x[y]

OctaveTranspositionMapping.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

OctaveTranspositionMapping.**__eq__**()
x.__eq__(y) <==> x==y

OctaveTranspositionMapping.**__ge__**()
x.__ge__(y) <==> x>=y

OctaveTranspositionMapping.**__getitem__**()
x.__getitem__(y) <==> x[y]

OctaveTranspositionMapping.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

OctaveTranspositionMapping.**__gt__**()
x.__gt__(y) <==> x>y

OctaveTranspositionMapping.**__iadd__**()
x.__iadd__(y) <==> x+=y

OctaveTranspositionMapping.**__imul__**()
x.__imul__(y) <==> x*=y

OctaveTranspositionMapping.**__iter__**() <==> *iter(x)*

OctaveTranspositionMapping.**__le__**()
x.__le__(y) <==> x<=y

OctaveTranspositionMapping.**__len__**() <==> *len(x)*

OctaveTranspositionMapping.**__lt__**()
x.__lt__(y) <==> x<y

OctaveTranspositionMapping.**__mul__**()
x.__mul__(n) <==> x*n

OctaveTranspositionMapping.**__ne__**()
x.__ne__(y) <==> x!=y

OctaveTranspositionMapping.**__repr__**()

OctaveTranspositionMapping.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

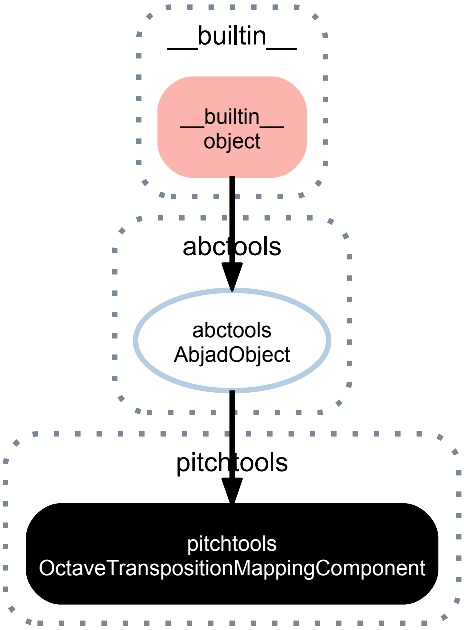
OctaveTranspositionMapping.**__rmul__**()
x.__rmul__(n) <==> n*x

OctaveTranspositionMapping.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

OctaveTranspositionMapping.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

21.2.51 pitchtools.OctaveTranspositionMappingComponent



class `pitchtools.OctaveTranspositionMappingComponent(*args)`

New in version 2.8. Octave transposition mapping component:

```
>>> pitchtools.OctaveTranspositionMappingComponent('[A0, C8]', 15)
OctaveTranspositionMappingComponent('[A0, C8]', 15)
```

Initialize from input parameters separately, from a pair, from a string or from another mapping component.

Model `pitchtools.transpose_chromatic_pitch_number_by_octave_transposition_mapping` input part. (See the docs for that function.)

Octave transposition mapping components are mutable.

Read-only properties

`OctaveTranspositionMappingComponent.storage_format`

Storage format of Abjad object.

Return string.

Read/write properties

`OctaveTranspositionMappingComponent.source_pitch_range`

Read / write source pitch range:

```
>>> mapping_component = pitchtools.OctaveTranspositionMappingComponent(
...     '[A0, C8]', 15)
>>> mapping_component.source_pitch_range
PitchRange('[A0, C8]')
```

Return pitch range or none.

`OctaveTranspositionMappingComponent.target_octave_start_pitch`

Read / write target octave start pitch:

```
>>> mapping_component = pitchtools.OctaveTranspositionMappingComponent(
...     '[A0, C8]', 15)
>>> mapping_component.target_octave_start_pitch
NumberedChromaticPitch(15)
```

Return numbered chromatic pitch or none.

Special methods

OctaveTranspositionMappingComponent.**__eq__**(*expr*)

OctaveTranspositionMappingComponent.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OctaveTranspositionMappingComponent.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

OctaveTranspositionMappingComponent.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

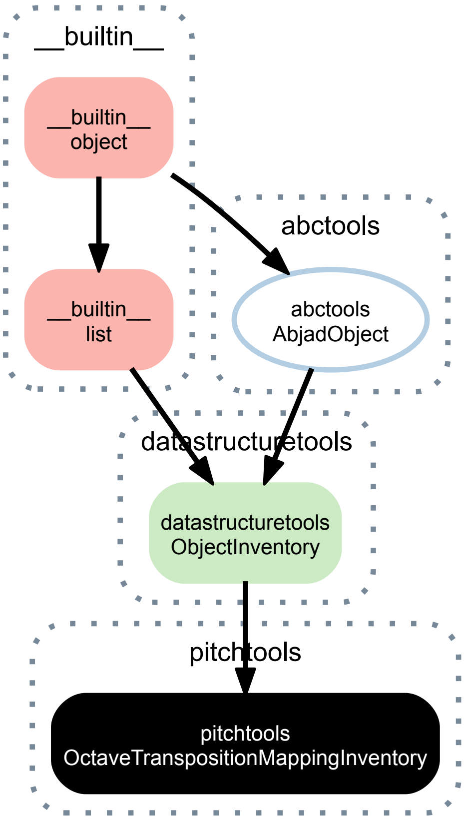
OctaveTranspositionMappingComponent.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OctaveTranspositionMappingComponent.**__ne__**(*expr*)

OctaveTranspositionMappingComponent.**__repr__**()

21.2.52 pitchtools.OctaveTranspositionMappingInventory



class pitchtools.**OctaveTranspositionMappingInventory**(*tokens=None*, *name=None*)
New in version 2.8. Model of an ordered list of octave transposition mappings:

```
>>> mapping_1 = pitchtools.OctaveTranspositionMapping([('[A0, C4)', 15), ('[C4, C8)', 27)])
>>> mapping_2 = pitchtools.OctaveTranspositionMapping([('[A0, C8]', -18)])
>>> inventory = pitchtools.OctaveTranspositionMappingInventory([mapping_1, mapping_2])
```

```
>>> z(inventory)
pitchtools.OctaveTranspositionMappingInventory([
    pitchtools.OctaveTranspositionMapping([
        pitchtools.OctaveTranspositionMappingComponent(
            pitchtools.PitchRange(
                '[A0, C4)'
                ),
            pitchtools.NumberedChromaticPitch(
                15
                )
            ),
        pitchtools.OctaveTranspositionMappingComponent(
            pitchtools.PitchRange(
                '[C4, C8)'
                ),
            pitchtools.NumberedChromaticPitch(
                27
                )
            )
        ]),
    pitchtools.OctaveTranspositionMapping([
        pitchtools.OctaveTranspositionMappingComponent(
            pitchtools.PitchRange(
                '[A0, C8]'
                ),
            pitchtools.NumberedChromaticPitch(
                -18
                )
            )
        ])
    ])
```

Octave transposition mapping inventories implement list interface and are mutable.

Read-only properties

OctaveTranspositionMappingInventory.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

OctaveTranspositionMappingInventory.**name**
Read / write name of inventory.

Methods

OctaveTranspositionMappingInventory.**append**(*token*)
Change *token* to item and append.

OctaveTranspositionMappingInventory.**count**(*value*) → integer – return number of occurrences of value

OctaveTranspositionMappingInventory.**extend**(*tokens*)
Change *tokens* to items and extend.

OctaveTranspositionMappingInventory.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`OctaveTranspositionMappingInventory.`**`insert`**`()`
L.insert(index, object) – insert object before index

`OctaveTranspositionMappingInventory.`**`pop`**`([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`OctaveTranspositionMappingInventory.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`OctaveTranspositionMappingInventory.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`OctaveTranspositionMappingInventory.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`OctaveTranspositionMappingInventory.`**`__add__`**`()`
x.__add__(y) <==> x+y

`OctaveTranspositionMappingInventory.`**`__contains__`**`(token)`

`OctaveTranspositionMappingInventory.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`OctaveTranspositionMappingInventory.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`OctaveTranspositionMappingInventory.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`OctaveTranspositionMappingInventory.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`OctaveTranspositionMappingInventory.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`OctaveTranspositionMappingInventory.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`OctaveTranspositionMappingInventory.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`OctaveTranspositionMappingInventory.`**`__iadd__`**`()`
x.__iadd__(y) <==> x+=y

`OctaveTranspositionMappingInventory.`**`__imul__`**`()`
x.__imul__(y) <==> x*=y

`OctaveTranspositionMappingInventory.`**`__iter__`**`()` *<==> iter(x)*

`OctaveTranspositionMappingInventory.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`OctaveTranspositionMappingInventory.`**`__len__`**`()` *<==> len(x)*

`OctaveTranspositionMappingInventory.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`OctaveTranspositionMappingInventory.`**`__mul__`**`()`
x.__mul__(n) <==> x*n

`OctaveTranspositionMappingInventory.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

OctaveTranspositionMappingInventory.**__repr__**()

OctaveTranspositionMappingInventory.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

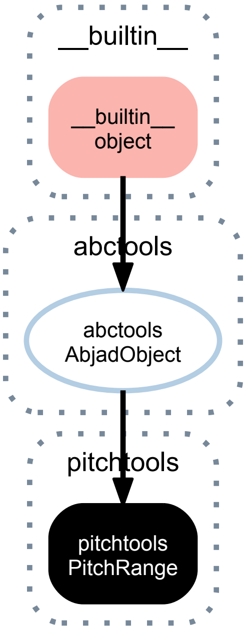
OctaveTranspositionMappingInventory.**__rmul__**()
x.__rmul__(n) <==> n*x

OctaveTranspositionMappingInventory.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

OctaveTranspositionMappingInventory.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

21.2.53 pitchtools.PitchRange



class pitchtools.**PitchRange**(**args*, ***kwargs*)
New in version 2.0. Abjad model of pitch range:

```
>>> pitchtools.PitchRange(-12, 36)
PitchRange('[C3, C7]')
```

Initalize from pitch numbers, pitch names, pitch instances, one-line reprs or other pitch range objects.

Pitch ranges implement equality testing against other pitch ranges.

Pitch ranges test less than, greater than, less-equal and greater-equal against pitches.

Pitch ranges do not sort relative to other pitch ranges.

Pitch ranges are immutable.

Read-only properties

PitchRange.**one_line_named_chromatic_pitch_repr**
Read-only one-line named chromatic pitch repr of pitch of range:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.one_line_named_chromatic_pitch_repr
'[C3, C7]'
```

Return string.

PitchRange.**one_line_numbered_chromatic_pitch_repr**
Read-only one-line numbered chromatic pitch repr of pitch of range:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.one_line_numbered_chromatic_pitch_repr
'[-12, 36]'
```

Return string.

PitchRange.**pitch_range_name**
New in version 2.7. Read-only name of pitch range:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36, pitch_range_name='four-octave range')
>>> pitch_range.pitch_range_name
'four-octave range'
```

Return string or none.

PitchRange.**pitch_range_name_markup**
New in version 2.7. Read-only markup of pitch range name:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36,
...     pitch_range_name_markup=markuptools.Markup('four-octave range'))
>>> pitch_range.pitch_range_name_markup
Markup(('four-octave range',))
```

Default to *pitch_range_name* when *pitch_range_name_markup* not set explicitly.

Return markup or none.

PitchRange.**start_pitch**
Read-only start pitch of range:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.start_pitch
NamedChromaticPitch('c')
```

Return pitch.

PitchRange.**start_pitch_is_included_in_range**
Read-only boolean true when start pitch is included in range. Otherwise false:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.start_pitch_is_included_in_range
True
```

Return boolean.

PitchRange.**stop_pitch**
Read-only stop pitch of range:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.stop_pitch
NamedChromaticPitch("c''''")
```

Return pitch.

PitchRange.**stop_pitch_is_included_in_range**
Read-only boolean true when stop pitch is included in range. Otherwise false:

```
>>> pitch_range = pitchtools.PitchRange(-12, 36)
>>> pitch_range.stop_pitch_is_included_in_range
True
```

Return boolean.

PitchRange.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

PitchRange.**__contains__**(*arg*)

PitchRange.**__eq__**(*arg*)

PitchRange.**__ge__**(*arg*)

PitchRange.**__gt__**(*arg*)

PitchRange.**__le__**(*arg*)

PitchRange.**__lt__**(*arg*)

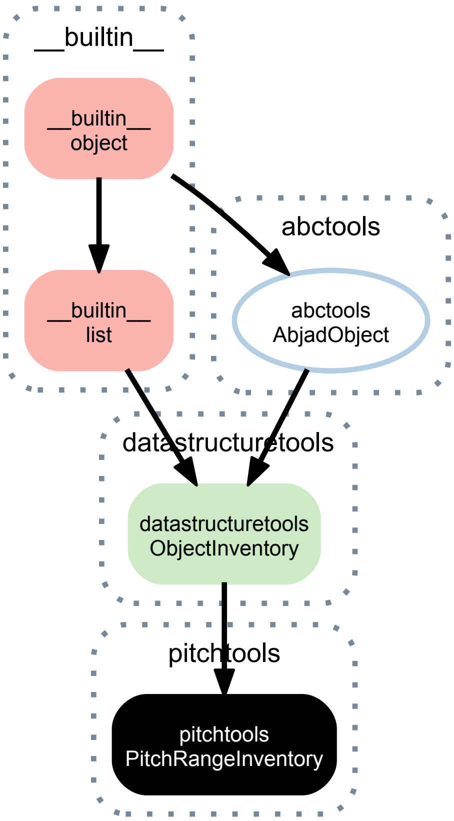
PitchRange.**__ne__**(*arg*)

PitchRange.**__repr__**()

Interpreter representation of Abjad object.

Return string.

21.2.54 pitchtools.PitchRangeInventory



class pitchtools.**PitchRangeInventory**(*tokens=None*, *name=None*)

New in version 2.7. Abjad model of an ordered list of pitch ranges:

```
>>> pitchtools.PitchRangeInventory(['[C3, C6]', '[C4, C6]'])
PitchRangeInventory([PitchRange('[C3, C6]'), PitchRange('[C4, C6]')])
```

Pitch range inventories implement list interface and are mutable.

Read-only properties

`PitchRangeInventory.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`PitchRangeInventory.`**`name`**
Read / write name of inventory.

Methods

`PitchRangeInventory.`**`append`**`(`*token*`)`
Change *token* to item and append.

`PitchRangeInventory.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`PitchRangeInventory.`**`extend`**`(`*tokens*`)`
Change *tokens* to items and extend.

`PitchRangeInventory.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`PitchRangeInventory.`**`insert`**`()`
L.insert(index, object) – insert object before index

`PitchRangeInventory.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`PitchRangeInventory.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`PitchRangeInventory.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`PitchRangeInventory.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`PitchRangeInventory.`**`__add__`**`()`
x.__add__(y) <==> x+y

`PitchRangeInventory.`**`__contains__`**`(`*token*`)`

`PitchRangeInventory.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`PitchRangeInventory.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`PitchRangeInventory.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`PitchRangeInventory.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`PitchRangeInventory.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

PitchRangeInventory.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

PitchRangeInventory.**__gt__**()
x.__gt__(y) <==> x>y

PitchRangeInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

PitchRangeInventory.**__imul__**()
x.__imul__(y) <==> x*=y

PitchRangeInventory.**__iter__**() <==> *iter(x)*

PitchRangeInventory.**__le__**()
x.__le__(y) <==> x<=y

PitchRangeInventory.**__len__**() <==> *len(x)*

PitchRangeInventory.**__lt__**()
x.__lt__(y) <==> x<y

PitchRangeInventory.**__mul__**()
x.__mul__(n) <==> x*n

PitchRangeInventory.**__ne__**()
x.__ne__(y) <==> x!=y

PitchRangeInventory.**__repr__**()

PitchRangeInventory.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

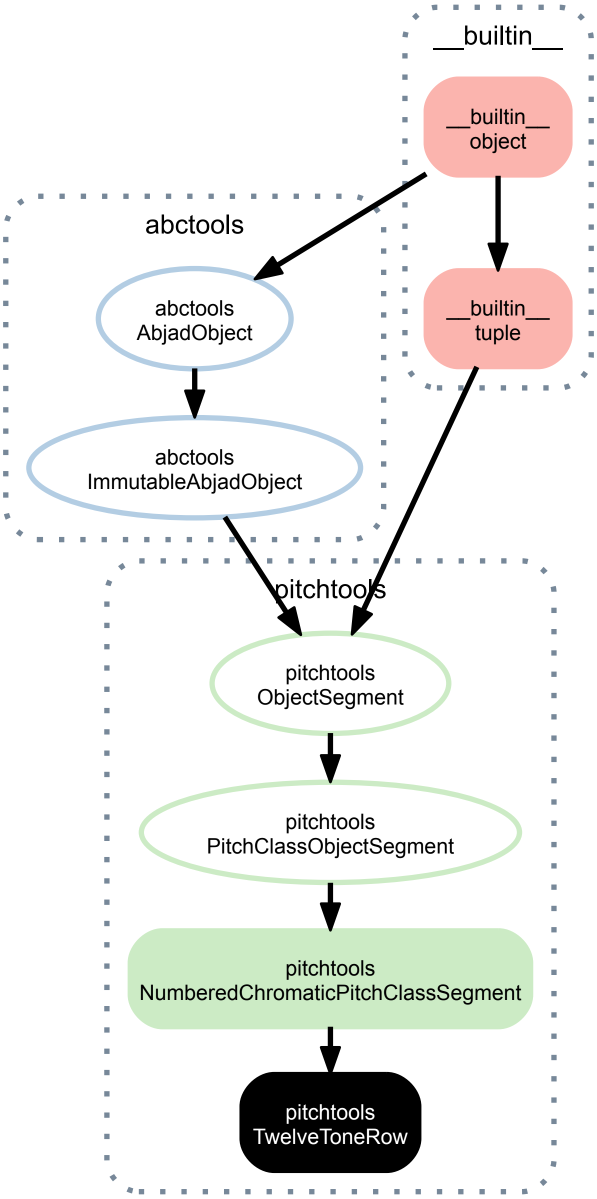
PitchRangeInventory.**__rmul__**()
x.__rmul__(n) <==> n*x

PitchRangeInventory.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

PitchRangeInventory.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

21.2.55 pitchtools.TwelveToneRow



class `pitchtools.`**`TwelveToneRow`**(**args*, ***kwargs*)

New in version 2.0. Abjad model of twelve-tone row:

```
>>> pitchtools.TwelveToneRow([0, 1, 11, 9, 3, 6, 7, 5, 4, 10, 2, 8])
TwelveToneRow([0, 1, 11, 9, 3, 6, 7, 5, 4, 10, 2, 8])
```

Twelve-tone rows validate pitch-classes at initialization.

Twelve-tone rows inherit canonical operators from numbered chromatic pitch-class segment.

Twelve-tone rows return numbered chromatic pitch-class segments on calls to getslice.

Twelve-tone rows are immutable.

Read-only properties

`TwelveToneRow.`**`inversion_equivalent_chromatic_interval_class_segment`**

Read-only inversion-equivalent chromatic interval-class segment:

```
>>> segment = pitchtools.NumberedChromaticPitchClassSegment([10, 10.5, 6, 7, 10.5, 7])
```

```
>>> segment.inversion_equivalent_chromatic_interval_class_segment
InversionEquivalentChromaticIntervalClassSegment(0.5, 4.5, 1, 3.5, 3.5)
```

Return inversion-equivalent chromatic interval-class segment.

`TwelveToneRow.`**`numbered_chromatic_pitch_class_set`**

Read-only numbered chromatic pitch-class set from numbered chromatic pitch-class segment:

```
>>> segment.numbered_chromatic_pitch_class_set
NumberedChromaticPitchClassSet([6, 7, 10, 10.5])
```

Return numbered chromatic pitch-class set.

`TwelveToneRow.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`TwelveToneRow.`**`alpha`**`()`

Morris alpha transform of numbered chromatic pitch-class segment:

```
>>> segment.alpha()
NumberedChromaticPitchClassSegment([11, 11.5, 7, 6, 11.5, 6])
```

Return numbered chromatic pitch-class segment.

`TwelveToneRow.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`TwelveToneRow.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.

Raises ValueError if the value is not present.

`TwelveToneRow.`**`invert`**`()`

Invert numbered chromatic pitch-class segment:

```
>>> segment.invert()
NumberedChromaticPitchClassSegment([2, 1.5, 6, 5, 1.5, 5])
```

Return numbered chromatic pitch-class segment.

`TwelveToneRow.`**`multiply`**`(`*n*`)`

Multiply numbered chromatic pitch-class segment by *n*:

```
>>> segment.multiply(5)
NumberedChromaticPitchClassSegment([2, 4.5, 6, 11, 4.5, 11])
```

Return numbered chromatic pitch-class segment.

`TwelveToneRow.`**`retrograde`**`()`

Retrograde of numbered chromatic pitch-class segment:

```
>>> segment.retrograde()
NumberedChromaticPitchClassSegment([7, 10.5, 7, 6, 10.5, 10])
```

Return numbered chromatic pitch-class segment.

`TwelveToneRow.`**`rotate`**`(`*n*`)`

Rotate numbered chromatic pitch-class segment:

```
>>> segment.rotate(1)
NumberedChromaticPitchClassSegment([7, 10, 10.5, 6, 7, 10.5])
```

Return numbered chromatic pitch-class segment.

TwelveToneRow.**transpose**(*n*)
Transpose numbered chromatic pitch-class segment:

```
>>> segment.transpose(10)
NumberedChromaticPitchClassSegment([8, 8.5, 4, 5, 8.5, 5])
```

Return numbered chromatic pitch-class segment.

Special methods

TwelveToneRow.**__add__**(*arg*)

TwelveToneRow.**__contains__**()
x.__contains__(y) <==> y in x

TwelveToneRow.**__copy__**()

TwelveToneRow.**__eq__**(*arg*)

TwelveToneRow.**__ge__**()
x.__ge__(y) <==> x>=y

TwelveToneRow.**__getitem__**()
x.__getitem__(y) <==> x[y]

TwelveToneRow.**__getslice__**(*start*, *stop*)

TwelveToneRow.**__gt__**()
x.__gt__(y) <==> x>y

TwelveToneRow.**__hash__**() <==> *hash(x)*

TwelveToneRow.**__iter__**() <==> *iter(x)*

TwelveToneRow.**__le__**()
x.__le__(y) <==> x<=y

TwelveToneRow.**__len__**() <==> *len(x)*

TwelveToneRow.**__lt__**()
x.__lt__(y) <==> x<y

TwelveToneRow.**__mul__**(*n*)

TwelveToneRow.**__ne__**(*arg*)

TwelveToneRow.**__repr__**()

TwelveToneRow.**__rmul__**(*n*)

TwelveToneRow.**__str__**()

21.3 Functions

21.3.1 pitchtools.all_are_chromatic_pitch_class_name_octave_number_pairs

pitchtools.**all_are_chromatic_pitch_class_name_octave_number_pairs**(*expr*)
New in version 1.1. True when all elements of *expr* are pitch tokens. Otherwise false:

```
>>> pitchtools.all_are_chromatic_pitch_class_name_octave_number_pairs(
... [('c', 4), ('d', 4), pitchtools.NamedChromaticPitch('e', 4)])
True
```

Return boolean.

21.3.2 pitchtools.all_are_named_chromatic_pitch_tokens

pitchtools.**all_are_named_chromatic_pitch_tokens**(*expr*)

New in version 2.6. True when *expr* is a sequence of named chromatic pitch tokens:

```
>>> named_chromatic_pitch_tokens = [('c', 4), pitchtools.NamedChromaticPitch("a'")]
```

```
>>> pitchtools.all_are_named_chromatic_pitch_tokens(named_chromatic_pitch_tokens)
True
```

True when *expr* is an empty sequence:

```
>>> pitchtools.all_are_named_chromatic_pitch_tokens([])
True
```

Otherwise false:

```
>>> pitchtools.all_are_named_chromatic_pitch_tokens('foo')
False
```

Return boolean.

21.3.3 pitchtools.alphabetic_accidental_abbreviation_to_symbolic_accidental_string

pitchtools.**alphabetic_accidental_abbreviation_to_symbolic_accidental_string**(*alphabetic_accid*

New in version 2.5. Change *alphabetic_accidental_abbreviation* to symbolic accidental string:

```
>>> pitchtools.alphabetic_accidental_abbreviation_to_symbolic_accidental_string('tqs')
'#+'
```

None when *alphabetic_accidental_abbreviation* is not a valid alphabetic accidental abbreviation.

Return string or none.

21.3.4 pitchtools.apply_accidental_to_named_chromatic_pitch

pitchtools.**apply_accidental_to_named_chromatic_pitch**(*named_chromatic_pitch*, *accidental=None*)

New in version 2.0. Apply *accidental* to *named_chromatic_pitch*:

```
>>> pitch = pitchtools.NamedChromaticPitch("cs''")
>>> pitchtools.apply_accidental_to_named_chromatic_pitch(pitch, 'f')
NamedChromaticPitch("c''")
```

Return new named pitch.

21.3.5 pitchtools.calculate_harmonic_chromatic_interval

pitchtools.**calculate_harmonic_chromatic_interval**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic chromatic interval from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_chromatic_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicChromaticInterval(14)
```

Return harmonic chromatic interval.

21.3.6 pitchtools.calculate_harmonic_chromatic_interval_class

pitchtools.**calculate_harmonic_chromatic_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic chromatic interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_chromatic_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicChromaticIntervalClass(2)
```

Return harmonic chromatic interval-class.

21.3.7 pitchtools.calculate_harmonic_counterpoint_interval

pitchtools.**calculate_harmonic_counterpoint_interval**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic counterpoint interval *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_counterpoint_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicCounterpointInterval(9)
```

Return harmonic counterpoint interval-class.

21.3.8 pitchtools.calculate_harmonic_counterpoint_interval_class

pitchtools.**calculate_harmonic_counterpoint_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic counterpoint interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_counterpoint_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicCounterpointIntervalClass(2)
```

Return harmonic counterpoint interval-class.

21.3.9 pitchtools.calculate_harmonic_diatonic_interval

pitchtools.**calculate_harmonic_diatonic_interval**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic diatonic interval from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_diatonic_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicDiatonicInterval('M9')
```

Return harmonic diatonic interval.

21.3.10 pitchtools.calculate_harmonic_diatonic_interval_class

pitchtools.**calculate_harmonic_diatonic_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate harmonic diatonic interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_harmonic_diatonic_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
HarmonicDiatonicIntervalClass('M2')
```

Return harmonic diatonic interval-class.

21.3.11 pitchtools.calculate_melodic_chromatic_interval

pitchtools.**calculate_melodic_chromatic_interval**(*pitch_carrier_1*, *pitch_carrier_2*)
New in version 2.0. Calculate melodic chromatic interval from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_chromatic_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicChromaticInterval(+14)
```

Return melodic chromatic interval.

21.3.12 pitchtools.calculate_melodic_chromatic_interval_class

pitchtools.**calculate_melodic_chromatic_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)
New in version 2.0. Calculate melodic chromatic interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_chromatic_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicChromaticIntervalClass(+2)
```

Return melodic chromatic interval-class.

21.3.13 pitchtools.calculate_melodic_counterpoint_interval

pitchtools.**calculate_melodic_counterpoint_interval**(*pitch_carrier_1*, *pitch_carrier_2*)
New in version 2.0. Calculate melodic counterpoint interval *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_counterpoint_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicCounterpointInterval(+9)
```

Return melodic counterpoint interval.

21.3.14 pitchtools.calculate_melodic_counterpoint_interval_class

pitchtools.**calculate_melodic_counterpoint_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)
New in version 2.0. Calculate melodic counterpoint interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_counterpoint_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicCounterpointIntervalClass(+2)
```

Return melodic counterpoint interval-class.

21.3.15 pitchtools.calculate_melodic_diatonic_interval

pitchtools.**calculate_melodic_diatonic_interval**(*pitch_carrier_1*, *pitch_carrier_2*)
New in version 2.0. Calculate melodic diatonic interval from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_diatonic_interval(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicDiatonicInterval('+M9')
```

Return melodic diatonic interval.

21.3.16 pitchtools.calculate_melodic_diatonic_interval_class

pitchtools.**calculate_melodic_diatonic_interval_class**(*pitch_carrier_1*, *pitch_carrier_2*)

New in version 2.0. Calculate melodic diatonic interval-class from *pitch_carrier_1* to *pitch_carrier_2*:

```
>>> pitchtools.calculate_melodic_diatonic_interval_class(
... pitchtools.NamedChromaticPitch(-2), pitchtools.NamedChromaticPitch(12))
MelodicDiatonicIntervalClass('+M2')
```

Return melodic diatonic interval-class.

21.3.17 pitchtools.chromatic_pitch_class_name_to_chromatic_pitch_class_number

pitchtools.**chromatic_pitch_class_name_to_chromatic_pitch_class_number**(*chromatic_pitch_class_na*

New in version 2.0. Change *chromatic_pitch_class_name* to chromatic pitch-class number:

```
>>> pitchtools.chromatic_pitch_class_name_to_chromatic_pitch_class_number('cs')
1
```

Return chromatic pitch-class number.

21.3.18 pitchtools.chromatic_pitch_class_name_to_diatonic_pitch_class_name

pitchtools.**chromatic_pitch_class_name_to_diatonic_pitch_class_name**(*chromatic_pitch_class_name*)

New in version 2.0. Change *chromatic_pitch_class_name* to diatonic pitch-class name:

```
>>> pitchtools.chromatic_pitch_class_name_to_diatonic_pitch_class_name('cs')
'c'
```

Return string.

21.3.19 pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name

pitchtools.**chromatic_pitch_class_number_to_chromatic_pitch_class_name**(*chromatic_pitch_class_nu*

New in version 1.1. Change *chromatic_pitch_class_number* to chromatic pitch-class name:

```
>>> tmp = pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name
>>> for n in range(0, 13):
...     pc = n / 2.0
...     pitch_name = tmp(pc)
...     print '%s   %s' % (pc, pitch_name)
...
0.0   c
0.5   cqs
1.0   cs
1.5   dqf
2.0   d
2.5   dqs
3.0   ef
3.5   eqf
4.0   e
4.5   eqs
5.0   f
5.5   fqs
6.0   fs
```

Return string.

21.3.20 pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_fla

pitchtools.**chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_flats**(*chromatic*

New in version 1.1. Change chromatic pitch-class number to chromatic pitch-class name with flats:

```
>>> tmp = pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_flats
```

```
>>> for n in range(13):
...     pc = n / 2.0
...     name = tmp(pc)
...     print '%s   %s' % (pc, name)
...
0.0   c
0.5   dtqf
1.0   df
1.5   dqf
2.0   d
2.5   etqf
3.0   ef
3.5   eqf
4.0   e
4.5   fqf
5.0   f
5.5   gtqf
6.0   gf
```

Return string.

21.3.21 pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_sh

pitchtools.**chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_sharps**(*chromat*

New in version 1.1. Change *chromatic_pitch_class_number* to chromatic pitch-class name with sharps:

```
>>> tmp = pitchtools.chromatic_pitch_class_number_to_chromatic_pitch_class_name_with_sharps
>>> for n in range(13):
...     pc = n / 2.0
...     name = tmp(pc)
...     print '%s   %s' % (pc, name)
...
0.0   c
0.5   cqs
1.0   cs
1.5   ctqs
2.0   d
2.5   dqs
3.0   ds
3.5   dtqs
4.0   e
4.5   eqs
5.0   f
5.5   fqs
6.0   fs
```

Return string.

21.3.22 pitchtools.chromatic_pitch_class_number_to_diatonic_pitch_class_number

pitchtools.**chromatic_pitch_class_number_to_diatonic_pitch_class_number**(*chromatic_pitch_class_n*

New in version 2.0. Change *chromatic_pitch_class_number* to diatonic pitch-class number:

```
>>> pitchtools.chromatic_pitch_class_number_to_diatonic_pitch_class_number(1)
0
```

Return integer.

21.3.23 pitchtools.chromatic_pitch_name_to_chromatic_pitch_class_name

pitchtools.**chromatic_pitch_name_to_chromatic_pitch_class_name**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_pitch_name* to chromatic pitch-class name:

```
>>> pitchtools.chromatic_pitch_name_to_chromatic_pitch_class_name("cs''")
'cs'
```

Return string.

21.3.24 pitchtools.chromatic_pitch_name_to_chromatic_pitch_class_number

pitchtools.**chromatic_pitch_name_to_chromatic_pitch_class_number**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_class_name* to chromatic pitch-class-number:

```
>>> pitchtools.chromatic_pitch_name_to_chromatic_pitch_class_number("cs''")
1
```

Return integer or float.

21.3.25 pitchtools.chromatic_pitch_name_to_chromatic_pitch_number

pitchtools.**chromatic_pitch_name_to_chromatic_pitch_number**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_pitch_name* to chromatic pitch number:

```
>>> pitchtools.chromatic_pitch_name_to_chromatic_pitch_number("cs''")
13
```

Return integer or float.

21.3.26 pitchtools.chromatic_pitch_name_to_diatonic_pitch_class_name

pitchtools.**chromatic_pitch_name_to_diatonic_pitch_class_name**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_pitch_name* to diatonic pitch name:

```
>>> pitchtools.chromatic_pitch_name_to_diatonic_pitch_class_name("cs''")
'c'
```

Return string.

21.3.27 pitchtools.chromatic_pitch_name_to_diatonic_pitch_class_number

pitchtools.**chromatic_pitch_name_to_diatonic_pitch_class_number**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_pitch_name* to diatonic pitch-class number:

```
>>> pitchtools.chromatic_pitch_name_to_diatonic_pitch_class_number("cs''")
0
```

Return integer.

21.3.28 pitchtools.chromatic_pitch_name_to_diatonic_pitch_name

pitchtools.**chromatic_pitch_name_to_diatonic_pitch_name**(*chromatic_pitch_name*)

New in version 2.0. Change *chromatic_pitch_name* to diatonic pitch name:

```
>>> pitchtools.chromatic_pitch_name_to_diatonic_pitch_name("cs''")
"c''"
```

Return string.

21.3.29 pitchtools.chromatic_pitch_name_to_diatonic_pitch_number

pitchtools.**chromatic_pitch_name_to_diatonic_pitch_number**(*chromatic_pitch_name*)
New in version 2.0. Change *chromatic_pitch_name* to diatonic pitch number:

```
>>> pitchtools.chromatic_pitch_name_to_diatonic_pitch_number("cs''")
7
```

Return integer.

21.3.30 pitchtools.chromatic_pitch_name_to_octave_number

pitchtools.**chromatic_pitch_name_to_octave_number**(*chromatic_pitch_name*)
New in version 2.0. Change *chromatic_pitch_name* to octave number:

```
>>> pitchtools.chromatic_pitch_name_to_octave_number('cs')
3
```

Return integer.

21.3.31 pitchtools.chromatic_pitch_names_string_to_named_chromatic_pitch_list

pitchtools.**chromatic_pitch_names_string_to_named_chromatic_pitch_list**(*chromatic_pitch_names_s*
New in version 2.0. Change *chromatic_pitch_names_string* to named chromatic pitch list:

```
>>> string = "cs, cs cs' cs''"
>>> result = pitchtools.chromatic_pitch_names_string_to_named_chromatic_pitch_list(string)
```

```
>>> for named_chromatic_pitch in result:
...     named_chromatic_pitch
...
NamedChromaticPitch('cs,')
NamedChromaticPitch('cs')
NamedChromaticPitch("cs'")
NamedChromaticPitch("cs''")
```

Return list of named chromatic pitches.

21.3.32 pitchtools.chromatic_pitch_number_and_accidental_semitones_to_octave_number

pitchtools.**chromatic_pitch_number_and_accidental_semitones_to_octave_number**(*chromatic_pitch_*
ac-
ci-
den-
tal_semitones)
New in version 1.1. Change *chromatic_pitch_number* and *accidental_semitones* to octave number:

```
>>> pitchtools.chromatic_pitch_number_and_accidental_semitones_to_octave_number(12, -2)
5
```

Return integer.

21.3.33 pitchtools.chromatic_pitch_number_to_chromatic_pitch_class_number

pitchtools.**chromatic_pitch_number_to_chromatic_pitch_class_number**(*chromatic_pitch_number*)
New in version 2.0. Change *chromatic_pitch_number* to chromatic pitch-class number:

```
>>> pitchtools.chromatic_pitch_number_to_chromatic_pitch_class_number(13)
1
```

Return integer or float.

21.3.34 pitchtools.chromatic_pitch_number_to_chromatic_pitch_name

pitchtools.**chromatic_pitch_number_to_chromatic_pitch_name**(*chromatic_pitch_number*, *accidental_spelling='mixed'*)

New in version 2.0. Change *chromatic_pitch_number* to chromatic pitch name:

```
>>> pitchtools.chromatic_pitch_number_to_chromatic_pitch_name(13)
"cs''"
```

Return string.

21.3.35 pitchtools.chromatic_pitch_number_to_chromatic_pitch_triple

pitchtools.**chromatic_pitch_number_to_chromatic_pitch_triple**(*chromatic_pitch_number*, *accidental_spelling='mixed'*)

Change *chromatic_pitch_number* to diatonic pitch-class name / alphabetic accidental abbreviation / octave number triple:

```
>>> pitchtools.chromatic_pitch_number_to_chromatic_pitch_triple(
... 13, accidental_spelling='sharps')
('c', Accidental('s'), 5)
```

Return tuple.

21.3.36 pitchtools.chromatic_pitch_number_to_diatonic_pitch_class_number

pitchtools.**chromatic_pitch_number_to_diatonic_pitch_class_number**(*chromatic_pitch_number*)

New in version 2.0. Change *chromatic_pitch_number* to diatonic pitch-class number:

```
>>> pitchtools.chromatic_pitch_number_to_diatonic_pitch_class_number(13)
0
```

Return integer.

21.3.37 pitchtools.chromatic_pitch_number_to_diatonic_pitch_number

pitchtools.**chromatic_pitch_number_to_diatonic_pitch_number**(*chromatic_pitch_number*)

New in version 2.0. Change *chromatic_pitch_number* to diatonic pitch number:

```
>>> pitchtools.chromatic_pitch_number_to_diatonic_pitch_number(13)
7
```

Return integer.

21.3.38 pitchtools.chromatic_pitch_number_to_octave_number

pitchtools.**chromatic_pitch_number_to_octave_number**(*chromatic_pitch_number*)

New in version 1.1. Change *chromatic_pitch_number* to octave number:

```
>>> pitchtools.chromatic_pitch_number_to_octave_number(13)
5
```

Return integer.

21.3.39 pitchtools.clef_and_staff_position_number_to_named_chromatic_pitch

pitchtools.**clef_and_staff_position_number_to_named_chromatic_pitch**(*clef*, *staff_position_number*)

New in version 2.0. Change *clef* and *staff_position_number* to named chromatic pitch:

```
>>> clef = contexttools.ClefMark('treble')
>>> for n in range(-6, 6):
...   pitch = pitchtools.clef_and_staff_position_number_to_named_chromatic_pitch(clef, n)
...   print '%s\t%s\t%s' % (clef.clef_name, n, pitch)
treble   -6 c'
treble   -5 d'
treble   -4 e'
treble   -3 f'
treble   -2 g'
treble   -1 a'
treble   0  b'
treble   1  c''
treble   2  d''
treble   3  e''
treble   4  f''
treble   5  g''
```

Return named chromatic pitch.

21.3.40 pitchtools.contains_subsegment

pitchtools.**contains_subsegment**(*chromatic_pitch_class_numbers*, *chromatic_pitch_numbers*)

New in version 1.1. True when *chromatic_pitch_numbers* contain *chromatic_pitch_class_numbers* as subsegment:

```
>>> pcs = [2, 7, 10]
>>> pitches = [6, 9, 12, 13, 14, 19, 22, 27, 28, 29, 32, 35]
>>> pitchtools.contains_subsegment(pcs, pitches)
True
```

Return boolean.

21.3.41 pitchtools.diatonic_pitch_class_name_to_chromatic_pitch_class_number

pitchtools.**diatonic_pitch_class_name_to_chromatic_pitch_class_number**(*diatonic_pitch_class_name*)

New in version 1.1. Change *diatonic_pitch_class_name* to chromatic pitch-class number:

```
>>> pitchtools.diatonic_pitch_class_name_to_chromatic_pitch_class_number('f')
5
```

Return integer.

21.3.42 pitchtools.diatonic_pitch_class_name_to_diatonic_pitch_class_number

pitchtools.**diatonic_pitch_class_name_to_diatonic_pitch_class_number**(*diatonic_pitch_class_name*)

New in version 2.0. Change *diatonic_pitch_class_name* to diatonic pitch-class number:

```
>>> pitchtools.diatonic_pitch_class_name_to_diatonic_pitch_class_number('c')
0
```

Return integer.

21.3.43 pitchtools.diatonic_pitch_class_number_to_chromatic_pitch_class_number

pitchtools.**diatonic_pitch_class_number_to_chromatic_pitch_class_number**(*diatonic_pitch_class_nu*
New in version 2.0. Change *diatonic_pitch_class_number* to chromatic pitch-class number:

```
>>> pitchtools.diatonic_pitch_class_number_to_chromatic_pitch_class_number(6)
11
```

Return nonnegative integer.

21.3.44 pitchtools.diatonic_pitch_class_number_to_diatonic_pitch_class_name

pitchtools.**diatonic_pitch_class_number_to_diatonic_pitch_class_name**(*diatonic_pitch_class_number*
New in version 2.0. Change *diatonic_pitch_class_number* to diatonic pitch-class name:

```
>>> pitchtools.diatonic_pitch_class_number_to_diatonic_pitch_class_name(0)
'c'
```

Return string.

21.3.45 pitchtools.diatonic_pitch_name_to_chromatic_pitch_class_name

pitchtools.**diatonic_pitch_name_to_chromatic_pitch_class_name**(*diatonic_pitch_name*)
New in version 2.0. Change *diatonic_pitch_name* to chromatic pitch-class name:

```
>>> pitchtools.diatonic_pitch_name_to_chromatic_pitch_class_name("c''")
'c'
```

Return string.

21.3.46 pitchtools.diatonic_pitch_name_to_chromatic_pitch_class_number

pitchtools.**diatonic_pitch_name_to_chromatic_pitch_class_number**(*diatonic_pitch_name*)
New in version 2.0. Change *diatonic_pitch_name* to chromatic pitch-class number:

```
>>> pitchtools.diatonic_pitch_name_to_chromatic_pitch_class_number("c''")
0
```

Return integer.

21.3.47 pitchtools.diatonic_pitch_name_to_chromatic_pitch_name

pitchtools.**diatonic_pitch_name_to_chromatic_pitch_name**(*diatonic_pitch_name*)
New in version 2.0. Change *diatonic_pitch_name* to chromatic pitch name:

```
>>> pitchtools.diatonic_pitch_name_to_chromatic_pitch_name("c''")
"c''"
```

Return string.

21.3.48 pitchtools.diatonic_pitch_name_to_chromatic_pitch_number

pitchtools.**diatonic_pitch_name_to_chromatic_pitch_number**(*diatonic_pitch_name*)
New in version 2.0. Change *diatonic_pitch_name* to chromatic pitch number:

```
>>> pitchtools.diatonic_pitch_name_to_chromatic_pitch_number("c''")
12
```

Return integer.

21.3.49 pitchtools.diatonic_pitch_name_to_diatonic_pitch_class_name

pitchtools.**diatonic_pitch_name_to_diatonic_pitch_class_name**(*diatonic_pitch_name*)

New in version 2.0. Change *diatonic_pitch_name* to diatonic pitch-class name:

```
>>> pitchtools.diatonic_pitch_name_to_diatonic_pitch_class_name("c''")
'c'
```

Return string.

21.3.50 pitchtools.diatonic_pitch_name_to_diatonic_pitch_class_number

pitchtools.**diatonic_pitch_name_to_diatonic_pitch_class_number**(*diatonic_pitch_name*)

New in version 2.0. Change *diatonic_pitch_name* to diatonic pitch-class number:

```
>>> pitchtools.diatonic_pitch_name_to_diatonic_pitch_class_number("c''")
0
```

Return integer.

21.3.51 pitchtools.diatonic_pitch_name_to_diatonic_pitch_number

pitchtools.**diatonic_pitch_name_to_diatonic_pitch_number**(*diatonic_pitch_name*)

New in version 2.0. Change *diatonic_pitch_name* to diatonic pitch number:

```
>>> pitchtools.diatonic_pitch_name_to_diatonic_pitch_number("c''")
7
```

Return integer.

21.3.52 pitchtools.diatonic_pitch_number_to_chromatic_pitch_number

pitchtools.**diatonic_pitch_number_to_chromatic_pitch_number**(*diatonic_pitch_number*)

New in version 2.0. Change *diatonic_pitch_number* to chromatic pitch number:

```
>>> pitchtools.diatonic_pitch_number_to_chromatic_pitch_number(7)
12
```

Return integer.

21.3.53 pitchtools.diatonic_pitch_number_to_diatonic_pitch_class_name

pitchtools.**diatonic_pitch_number_to_diatonic_pitch_class_name**(*diatonic_pitch_number*)

New in version 2.0. Change *diatonic_pitch_number* to diatonic pitch-class name:

```
>>> pitchtools.diatonic_pitch_number_to_diatonic_pitch_class_name(7)
'c'
```

Return string.

21.3.54 pitchtools.diatonic_pitch_number_to_diatonic_pitch_class_number

pitchtools.**diatonic_pitch_number_to_diatonic_pitch_class_number**(*diatonic_pitch_number*)

New in version 2.0. Change *diatonic_pitch_number* to diatonic pitch-class number:

```
>>> pitchtools.diatonic_pitch_number_to_diatonic_pitch_class_number(7)
0
```

Return nonnegative integer.

21.3.55 pitchtools.diatonic_pitch_number_to_diatonic_pitch_name

pitchtools.**diatonic_pitch_number_to_diatonic_pitch_name**(*diatonic_pitch_number*)
New in version 2.0. Change *diatonic_pitch_number* to diatonic pitch name:

```
>>> pitchtools.diatonic_pitch_number_to_diatonic_pitch_name(7)
"c''"
```

Return string.

21.3.56 pitchtools.expr_has_duplicate_named_chromatic_pitch

pitchtools.**expr_has_duplicate_named_chromatic_pitch**(*expr*)
New in version 2.0. True when *expr* has duplicate named chromatic pitch. Otherwise false:

```
>>> chord = Chord([13, 13, 14], (1, 4))
>>> pitchtools.expr_has_duplicate_named_chromatic_pitch(chord)
True
```

Return boolean.

21.3.57 pitchtools.expr_has_duplicate_numbered_chromatic_pitch_class

pitchtools.**expr_has_duplicate_numbered_chromatic_pitch_class**(*expr*)
New in version 2.0. True when *expr* has duplicate numbered chromatic pitch-class. Otherwise false:

```
>>> chord = Chord([1, 13, 14], (1, 4))
>>> pitchtools.expr_has_duplicate_numbered_chromatic_pitch_class(chord)
True
```

Return boolean.

21.3.58 pitchtools.expr_to_melodic_chromatic_interval_segment

pitchtools.**expr_to_melodic_chromatic_interval_segment**(*expr*)
New in version 2.0. Change *expr* to melodic chromatic interval segment:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> pitchtools.expr_to_melodic_chromatic_interval_segment(staff)
MelodicChromaticIntervalSegment(+2, +2, +1, +2, +2, +2, +1)
```

Return melodic chromatic interval segment.

21.3.59 pitchtools.get_named_chromatic_pitch_from_pitch_carrier

pitchtools.**get_named_chromatic_pitch_from_pitch_carrier**(*pitch_carrier*)
New in version 1.1. Get named chromatic pitch from *pitch_carrier*:

```
>>> pitch = pitchtools.NamedChromaticPitch('df', 5)
>>> pitch
NamedChromaticPitch("df''")
>>> pitchtools.get_named_chromatic_pitch_from_pitch_carrier(pitch)
NamedChromaticPitch("df''")
```

```
>>> note = Note(('df', 5), (1, 4))
>>> note
Note("df''4")
>>> pitchtools.get_named_chromatic_pitch_from_pitch_carrier(note)
NamedChromaticPitch("df''")
```

```
>>> note = Note(('df', 5), (1, 4))
>>> note.note_head
NoteHead("df''")
>>> pitchtools.get_named_chromatic_pitch_from_pitch_carrier(note.note_head)
NamedChromaticPitch("df''")
```

```
>>> chord = Chord([('df', 5)], (1, 4))
>>> chord
Chord("<df''>4")
>>> pitchtools.get_named_chromatic_pitch_from_pitch_carrier(chord)
NamedChromaticPitch("df''")
```

```
>>> pitchtools.get_named_chromatic_pitch_from_pitch_carrier(13)
NamedChromaticPitch("cs''")
```

Raise missing pitch error when *pitch_carrier* carries no pitch.

Raise extra pitch error when *pitch_carrier* carries more than one pitch.

Return named chromatic pitch.

21.3.60 pitchtools.get_numbered_chromatic_pitch_class_from_pitch_carrier

pitchtools.**get_numbered_chromatic_pitch_class_from_pitch_carrier**(*pitch_carrier*)
New in version 2.0. Get numbered chromatic pitch-class from *pitch_carrier*:

```
>>> note = Note("cs'4")
>>> pitchtools.get_numbered_chromatic_pitch_class_from_pitch_carrier(note)
NumberedChromaticPitchClass(1)
```

Raise missing pitch error on empty chords.

Raise extra pitch error on many-note chords.

Return numbered chromatic pitch-class.

21.3.61 pitchtools.harmonic_chromatic_interval_class_number_dictionary

pitchtools.**harmonic_chromatic_interval_class_number_dictionary**(*pitches*)
New in version 1.1. Change named chromatic pitches to harmonic chromatic interval-class number dictionary:

```
>>> chord = Chord([0, 2, 11], (1, 4))
>>> vector = pitchtools.harmonic_chromatic_interval_class_number_dictionary(
... chord.written_pitches)
>>> vector
{0: 0, 1: 0, 2: 1, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 1, 10: 0, 11: 1}
```

Return dictionary.

21.3.62 pitchtools.insert_and_transpose_nested_subruns_in_chromatic_pitch_class_numbe

pitchtools.**insert_and_transpose_nested_subruns_in_chromatic_pitch_class_number_list**(*notes*, *sub-run_i*

New in version 1.1. Insert and transpose nested subruns in *chromatic_pitch_class_number_list* according to *subrun_indicators*:

```
>>> notes = [Note(p, (1, 4)) for p in [0, 2, 7, 9, 5, 11, 4]]
>>> subrun_indicators = [(0, [2, 4]), (4, [3, 1])]
>>> pitchtools.insert_and_transpose_nested_subruns_in_chromatic_pitch_class_number_list(
... notes, subrun_indicators)

>>> t = []
```

```
>>> for x in notes:
...   try:
...        t.append(x.written_pitch.chromatic_pitch_number)
...   except AttributeError:
...        t.append([y.written_pitch.chromatic_pitch_number for y in x])

>>> t
[0, [5, 7], 2, [4, 0, 6, 11], 7, 9, 5, [10, 6, 8], 11, [7], 4]
```

Set *subrun_indicators* to a list of zero or more `(index, length_list)` pairs.

For each `(index, length_list)` pair in *subrun_indicators* the function will read *index* mod `len(notes)` and insert a subrun of length `length_list[0]` immediately after `notes[index]`, a subrun of length `length_list[1]` immediately after `notes[index+1]`, and, in general, a subrun of `length_list[i]` immediately after `notes[index+i]`, for `i < length(length_list)`.

New subruns are wrapped with lists. These wrapper lists are designed to allow inspection of the structural changes to *notes* immediately after the function returns. For this reason most calls to this function will be followed by `notes = sequencetools.flatten_sequence(notes)`:

```
>>> for note in notes: note
...
Note("c'4")
[Note("f'4"), Note("g'4")]
Note("d'4")
[Note("e'4"), Note("c'4"), Note("fs'4"), Note("b'4")]
Note("g'4")
Note("a'4")
Note("f'4")
[Note("bf'4"), Note("fs'4"), Note("af'4")]
Note("b'4")
[Note("g'4")]
Note("e'4")
```

This function is designed to work on a built-in Python list of notes. This function is **not** designed to work on Abjad voices, staves or other containers because the function currently implements no spanner-handling. That is, this function is designed to be used during precomposition when other, similar abstract pitch transforms may be common.

Return list of integers and / or floats.

21.3.63 pitchtools.instantiate_pitch_and_interval_test_collection

`pitchtools.instantiate_pitch_and_interval_test_collection()`

New in version 2.0. Instantiate pitch and interval test collection:

```
>>> for x in pitchtools.instantiate_pitch_and_interval_test_collection(): x
...
HarmonicChromaticInterval(1)
HarmonicChromaticIntervalClass(1)
HarmonicCounterpointInterval(1)
HarmonicCounterpointIntervalClass(1)
HarmonicDiatonicInterval('M2')
HarmonicDiatonicIntervalClass('M2')
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentDiatonicIntervalClass('M2')
MelodicChromaticInterval(+1)
MelodicChromaticIntervalClass(+1)
MelodicCounterpointInterval(1)
MelodicCounterpointIntervalClass(+1)
MelodicDiatonicInterval('+M2')
MelodicDiatonicIntervalClass('+M2')
NamedChromaticPitch('c')
NamedChromaticPitchClass('c')
NamedDiatonicPitch('c')
NamedDiatonicPitchClass('c')
NumberedChromaticPitch(1)
NumberedChromaticPitchClass(1)
```

```
NumberedDiatonicPitch(1)
NumberedDiatonicPitchClass(1)
```

Use to test pitch and interval interface consistency.

Return list.

21.3.64 pitchtools.inventory_aggregate_subsets

pitchtools.**inventory_aggregate_subsets**()

New in version 2.0. Inventory aggregate subsets:

```
>>> U_star = pitchtools.inventory_aggregate_subsets()
>>> len(U_star)
4096
>>> for pcset in U_star[:20]:
...   pcset
NumberedChromaticPitchClassSet([])
NumberedChromaticPitchClassSet([0])
NumberedChromaticPitchClassSet([1])
NumberedChromaticPitchClassSet([0, 1])
NumberedChromaticPitchClassSet([2])
NumberedChromaticPitchClassSet([0, 2])
NumberedChromaticPitchClassSet([1, 2])
NumberedChromaticPitchClassSet([0, 1, 2])
NumberedChromaticPitchClassSet([3])
NumberedChromaticPitchClassSet([0, 3])
NumberedChromaticPitchClassSet([1, 3])
NumberedChromaticPitchClassSet([0, 1, 3])
NumberedChromaticPitchClassSet([2, 3])
NumberedChromaticPitchClassSet([0, 2, 3])
NumberedChromaticPitchClassSet([1, 2, 3])
NumberedChromaticPitchClassSet([0, 1, 2, 3])
NumberedChromaticPitchClassSet([4])
NumberedChromaticPitchClassSet([0, 4])
NumberedChromaticPitchClassSet([1, 4])
NumberedChromaticPitchClassSet([0, 1, 4])
```

There are 4096 subsets of the aggregate.

This is `U*` in [Morris 1987].

Return list of numbered chromatic pitch-class sets.

21.3.65 pitchtools.inventory_inversion_equivalent_diatonic_interval_classes

pitchtools.**inventory_inversion_equivalent_diatonic_interval_classes**()

New in version 2.0. Inventory inversion-equivalent diatonic interval-classes:

```
>>> for dic in pitchtools.inventory_inversion_equivalent_diatonic_interval_classes():
...     dic
...
InversionEquivalentDiatonicIntervalClass('P1')
InversionEquivalentDiatonicIntervalClass('aug1')
InversionEquivalentDiatonicIntervalClass('m2')
InversionEquivalentDiatonicIntervalClass('M2')
InversionEquivalentDiatonicIntervalClass('aug2')
InversionEquivalentDiatonicIntervalClass('dim3')
InversionEquivalentDiatonicIntervalClass('m3')
InversionEquivalentDiatonicIntervalClass('M3')
InversionEquivalentDiatonicIntervalClass('dim4')
InversionEquivalentDiatonicIntervalClass('P4')
InversionEquivalentDiatonicIntervalClass('aug4')
```

There are 11 inversion-equivalent diatonic interval-classes.

It is an open question as to whether octaves should be included.

Return list of inversion-equivalent diatonic interval-classes.

21.3.66 pitchtools.inversion_equivalent_chromatic_interval_class_number_dictionary

pitchtools.**inversion_equivalent_chromatic_interval_class_number_dictionary**(*pitches*)
New in version 1.1. Change named chromatic *pitches* to inversion-equivalent chromatic interval-class number dictionary:

```
>>> chord = Chord("<c' d' b''>4")
>>> vector = pitchtools.inversion_equivalent_chromatic_interval_class_number_dictionary(
... chord.written_pitches)
>>> for i in range(7):
...     print '\t%s\t%s' % (i, vector[i])
...
    0   0
    1   1
    2   1
    3   1
    4   0
    5   0
    6   0
```

Return dictionary.

21.3.67 pitchtools.is_alphabetic_accidental_abbreviation

pitchtools.**is_alphabetic_accidental_abbreviation**(*expr*)
New in version 2.0. True when *expr* is an alphabetic accidental abbrevation. Otherwise false:

```
>>> pitchtools.is_alphabetic_accidental_abbreviation('tqs')
True
```

The regex `^([s]{1,2}|[f]{1,2}|t?q?[fs])!?$` underlies this predicate.

Return boolean.

21.3.68 pitchtools.is_chromatic_pitch_class_name

pitchtools.**is_chromatic_pitch_class_name**(*expr*)
New in version 2.0. True when *expr* is a chromatic pitch-class name. Otherwise false:

```
>>> pitchtools.is_chromatic_pitch_class_name('fs')
True
```

The regex `^([a-g,A-G])(([s]{1,2}|[f]{1,2}|t?q?[fs]|)!?)$` underlies this predicate.

Return boolean.

21.3.69 pitchtools.is_chromatic_pitch_class_name_octave_number_pair

pitchtools.**is_chromatic_pitch_class_name_octave_number_pair**(*expr*)
New in version 1.1. True when *arg* has the form of a chromatic pitch-class / octave number pair. Otherwise false:

```
>>> pitchtools.is_chromatic_pitch_class_name_octave_number_pair(('cs', 5))
True
```

Return boolean.

21.3.70 pitchtools.is_chromatic_pitch_class_number

pitchtools.**is_chromatic_pitch_class_number**(*expr*)

New in version 2.0. True *expr* is a chromatic pitch-class number. Otherwise false:

```
>>> pitchtools.is_chromatic_pitch_class_number(1)
True
```

The chromatic pitch-class numbers are equal to the set `[0, 0.5, ..., 11, 11.5]`.

Return boolean.

21.3.71 pitchtools.is_chromatic_pitch_name

pitchtools.**is_chromatic_pitch_name**(*expr*)

New in version 2.0. True *expr* is a chromatic pitch name. Otherwise false:

```
>>> pitchtools.is_chromatic_pitch_name('c,')
True
```

The regex `^([a-g,A-G])(([s]{1,2}|[f]{1,2}|t?q?[f,s]|)!?)(,+|'+|)$` underlies this predicate.

Return boolean.

21.3.72 pitchtools.is_chromatic_pitch_number

pitchtools.**is_chromatic_pitch_number**(*expr*)

New in version 2.0. True *expr* is a chromatic pitch number. Otherwise false:

```
>>> pitchtools.is_chromatic_pitch_number(13)
True
```

The chromatic pitch numbers are equal to the set of all integers in union with the set of all integers plus of minus `0.5`.

Return boolean.

21.3.73 pitchtools.is_diatonic_pitch_class_name

pitchtools.**is_diatonic_pitch_class_name**(*expr*)

New in version 2.0. True when *expr* is a diatonic pitch-class name. Otherwise false:

```
>>> pitchtools.is_diatonic_pitch_class_name('c')
True
```

The regex `^[a-g,A-G]$` underlies this predicate.

Return boolean.

21.3.74 pitchtools.is_diatonic_pitch_class_number

pitchtools.**is_diatonic_pitch_class_number**(*expr*)

New in version 2.0. True when *expr* is a diatonic pitch-class number. Otherwise false:

```
>>> pitchtools.is_diatonic_pitch_class_number(0)
True
```

The diatonic pitch-class numbers are equal to the set `[0, 1, 2, 3, 4, 5, 6]`.

Return boolean.

21.3.75 pitchtools.is_diatonic_pitch_name

`pitchtools.is_diatonic_pitch_name(expr)`

New in version 2.0. True when *expr* is a diatonic pitch name. Otherwise false:

```
>>> pitchtools.is_diatonic_pitch_name("c''")
True
```

The regex `(^[a-g,A-G])(,+|'+|)$` underlies this predicate.

Return boolean.

21.3.76 pitchtools.is_diatonic_pitch_number

`pitchtools.is_diatonic_pitch_number(expr)`

New in version 2.0. True when *expr* is a diatonic pitch number. Otherwise false:

```
>>> pitchtools.is_diatonic_pitch_number(7)
True
```

The diatonic pitch numbers are equal to the set of integers.

Return boolean.

21.3.77 pitchtools.is_diatonic_quality_abbreviation

`pitchtools.is_diatonic_quality_abbreviation(expr)`

New in version 2.0. True when *expr* is a diatonic quality abbreviation. Otherwise false:

```
>>> pitchtools.is_diatonic_quality_abbreviation('aug')
True
```

The regex `^M|m|P|aug|dim$` underlies this predicate.

Return boolean.

21.3.78 pitchtools.is_harmonic_diatonic_interval_abbreviation

`pitchtools.is_harmonic_diatonic_interval_abbreviation(expr)`

New in version 2.0. True when *expr* is a harmonic diatonic interval abbreviation. Otherwise false:

```
>>> pitchtools.is_harmonic_diatonic_interval_abbreviation('M9')
True
```

The regex `^(M|m|P|aug|dim)(\d+)$` underlies this predicate.

Return boolean.

21.3.79 pitchtools.is_melodic_diatonic_interval_abbreviation

`pitchtools.is_melodic_diatonic_interval_abbreviation(expr)`

New in version 2.0. True when *expr* is a melodic diatonic interval abbreviation. Otherwise false:

```
>>> pitchtools.is_melodic_diatonic_interval_abbreviation('+M9')
True
```

The regex `^([+,-]?)(M|m|P|aug|dim)(\d+)$` underlies this predicate.

Return boolean.

21.3.80 pitchtools.is_named_chromatic_pitch_token

pitchtools.**is_named_chromatic_pitch_token**(*pitch_token*)
New in version 1.1. True when *pitch_token* has the form of an Abjad pitch token. Otherwise false:

```
>>> pitchtools.is_named_chromatic_pitch_token(('c', 4))
True
```

Return boolean.

21.3.81 pitchtools.is_octave_tick_string

pitchtools.**is_octave_tick_string**(*expr*)
New in version 2.0. True when *expr* is an octave tick string. Otherwise false:

```
>>> pitchtools.is_octave_tick_string(',,,')
True
```

The regex `^,+|'+|$` underlies this predicate.

Return boolean.

21.3.82 pitchtools.is_pitch_carrier

pitchtools.**is_pitch_carrier**(*expr*)
New in version 1.1. True when *expr* is an Abjad pitch, note, note-head of chord instance. Otherwise false:

```
>>> note = Note("c'4")
>>> pitchtools.is_pitch_carrier(note)
True
```

Return boolean.

21.3.83 pitchtools.is_pitch_class_octave_number_string

pitchtools.**is_pitch_class_octave_number_string**(*expr*)
New in version 2.5. True when *expr* is a pitch-class / octave number string. Otherwise false:

```
>>> pitchtools.is_pitch_class_octave_number_string('C#2')
True
```

Quartertone accidentals are supported.

The regex `^([A-G])([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)([-]?[0-9]+)$` underlies this predicate.

Return boolean.

21.3.84 pitchtools.is_symbolic_accidental_string

pitchtools.**is_symbolic_accidental_string**(*expr*)
New in version 2.5. True when *expr* is a symbolic accidental string. Otherwise false:

```
>>> pitchtools.is_symbolic_accidental_string('#+')
True
```

True on empty string.

The regex `^([#]{1,2}|[b]{1,2}|[#]?[+]|[b]?[~]|)$` underlies this predicate.

Return boolean.

21.3.85 pitchtools.is_symbolic_pitch_range_string

pitchtools.**is_symbolic_pitch_range_string**(*expr*)

New in version 2.5. True when *expr* is a symbolic pitch range string. Otherwise false:

```
>>> pitchtools.is_symbolic_pitch_range_string('[A0, C8]')
True
```

The regex that underlies this predicate matches against two comma-separated pitch indicators enclosed in some combination of square brackets and round parentheses.

Return boolean.

21.3.86 pitchtools.iterate_named_chromatic_pitch_pairs_in_expr

pitchtools.**iterate_named_chromatic_pitch_pairs_in_expr**(*expr*)

New in version 2.0. Iterate left-to-right, top-to-bottom named chromatic pitch pairs in *expr*:

```
>>> score = Score([])
>>> notes = [Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'4")]
>>> score.append(Staff(notes))
>>> notes = [Note(x, (1, 4)) for x in [-12, -15, -17]]
>>> score.append(Staff(notes))
>>> contexttools.ClefMark('bass')(score[1])
ClefMark('bass')(Staff{3})
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
        g'4
    }
    \new Staff {
        \clef "bass"
        c4
        a,4
        g,4
    }
>>
```

```
>>> for pair in pitchtools.iterate_named_chromatic_pitch_pairs_in_expr(score):
...     pair
...
(NamedChromaticPitch("c'"), NamedChromaticPitch('c'))
(NamedChromaticPitch("c'"), NamedChromaticPitch("d'"))
(NamedChromaticPitch('c'), NamedChromaticPitch("d'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("e'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch('a,'))
(NamedChromaticPitch('c'), NamedChromaticPitch("e'"))
(NamedChromaticPitch('c'), NamedChromaticPitch('a,'))
(NamedChromaticPitch("e'"), NamedChromaticPitch('a,'))
(NamedChromaticPitch("e'"), NamedChromaticPitch("f'"))
(NamedChromaticPitch('a,'), NamedChromaticPitch("f'"))
(NamedChromaticPitch("f'"), NamedChromaticPitch("g'"))
(NamedChromaticPitch("f'"), NamedChromaticPitch('g,'))
(NamedChromaticPitch('a,'), NamedChromaticPitch("g'"))
(NamedChromaticPitch('a,'), NamedChromaticPitch('g,'))
(NamedChromaticPitch("g'"), NamedChromaticPitch('g,'))
```

Chords are handled correctly.

```
>>> chord_1 = Chord([0, 2, 4], (1, 4))
>>> chord_2 = Chord([17, 19], (1, 4))
>>> staff = Staff([chord_1, chord_2])
```

```
>>> f(staff)
\new Staff {
    <c' d' e'>4
    <f'' g''>4
}
```

```
>>> for pair in pitchtools.iterate_named_chromatic_pitch_pairs_in_expr(staff):
...   print pair
(NamedChromaticPitch("c'"), NamedChromaticPitch("d'"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("e'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("e'"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("f''"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("g''"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("f''"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("g''"))
(NamedChromaticPitch("e'"), NamedChromaticPitch("f''"))
(NamedChromaticPitch("e'"), NamedChromaticPitch("g''"))
(NamedChromaticPitch("f''"), NamedChromaticPitch("g''"))
```

Return generator.

21.3.87 pitchtools.list_chromatic_pitch_numbers_in_expr

pitchtools.**list_chromatic_pitch_numbers_in_expr**(*expr*)
New in version 2.0. List chromatic pitch numbers in *expr*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> pitchtools.list_chromatic_pitch_numbers_in_expr(tuplet)
(0, 2, 4)
```

Return tuple of zero or more numbers.

21.3.88 pitchtools.list_harmonic_chromatic_intervals_in_expr

pitchtools.**list_harmonic_chromatic_intervals_in_expr**(*expr*)
New in version 2.0. List harmonic chromatic intervals in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> for interval in sorted(pitchtools.list_harmonic_chromatic_intervals_in_expr(staff)):
...     interval
...
HarmonicChromaticInterval(1)
HarmonicChromaticInterval(2)
HarmonicChromaticInterval(2)
HarmonicChromaticInterval(3)
HarmonicChromaticInterval(4)
HarmonicChromaticInterval(5)
```

Return unordered set.

21.3.89 pitchtools.list_harmonic_diatonic_intervals_in_expr

pitchtools.**list_harmonic_diatonic_intervals_in_expr**(*expr*)
New in version 2.0. List harmonic diatonic intervals in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> for interval in sorted(pitchtools.list_harmonic_diatonic_intervals_in_expr(staff)):
...     interval
...
HarmonicDiatonicInterval('m2')
HarmonicDiatonicInterval('M2')
HarmonicDiatonicInterval('M2')
HarmonicDiatonicInterval('m3')
HarmonicDiatonicInterval('M3')
HarmonicDiatonicInterval('P4')
```

Return unordered set.

21.3.90 pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise

pitchtools.**list_inversion_equivalent_chromatic_interval_classes_pairwise**(*pitch_carriers*, *wrap=False*)

New in version 2.0. List inversion-equivalent chromatic interval-classes pairwise between *pitch_carriers*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
    b'8
    c''8
}
```

```
>>> result = pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise(
... staff, wrap=False)
```

```
>>> for x in result: x
...
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
```

```
>>> result = pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise(
... staff, wrap=True)
```

```
>>> for x in result: x
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(0)
```

```
>>> notes = staff.leaves
>>> notes = list(reversed(notes))
```

```
>>> result = pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise(
... notes, wrap=False)
```

```
>>> for x in result: x
...
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
```

```
>>> result = pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise(
... notes, wrap=True)
```

```
>>> for x in result: x
...
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(1)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(2)
InversionEquivalentChromaticIntervalClass(0)
```

When `wrap=False` do not return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

When `wrap=True` do return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

Return list.

21.3.91 pitchtools.list_melodic_chromatic_interval_numbers_pairwise

`pitchtools.`**`list_melodic_chromatic_interval_numbers_pairwise`**(*pitch_carriers*, *wrap=False*)

New in version 1.1. List melodic chromatic interval numbers pairwise between *pitch_carriers*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
    b'8
    c''8
}
```

```
>>> pitchtools.list_melodic_chromatic_interval_numbers_pairwise(
... staff)
[2, 2, 1, 2, 2, 2, 1]
```

```
>>> pitchtools.list_melodic_chromatic_interval_numbers_pairwise(
... staff, wrap=True)
[2, 2, 1, 2, 2, 2, 1, -12]
```

```
>>> notes = [
...     Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"),
...     Note("g'8"), Note("a'8"), Note("b'8"), Note("c''8")]
```

```
>>> notes.reverse()
```

```
>>> pitchtools.list_melodic_chromatic_interval_numbers_pairwise(
... notes)
[-1, -2, -2, -2, -1, -2, -2]
```

```
>>> pitchtools.list_melodic_chromatic_interval_numbers_pairwise(
... notes, wrap=True)
[-1, -2, -2, -2, -1, -2, -2, 12]
```

When `wrap = False` do not return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

When `wrap = True` do return `pitch_carriers[-1] - pitch_carriers[0]` as last in series.

Return list.

21.3.92 pitchtools.list_named_chromatic_pitches_in_expr

pitchtools.**list_named_chromatic_pitches_in_expr**(*expr*)

New in version 2.0. List named chromatic pitches in *expr*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> beam_spanner = beamtools.BeamSpanner(staff[:])
```

```
>>> for x in pitchtools.list_named_chromatic_pitches_in_expr(beam_spanner):
...     x
...
NamedChromaticPitch("c'")
NamedChromaticPitch("d'")
NamedChromaticPitch("e'")
NamedChromaticPitch("f'")
```

Return tuple.

21.3.93 pitchtools.list_numbered_chromatic_pitch_classes_in_expr

pitchtools.**list_numbered_chromatic_pitch_classes_in_expr**(*expr*)

New in version 2.0. List numbered chromatic pitch-classes in *expr*:

```
>>> chord = Chord("<cs'' d'' ef''>4")
```

```
>>> for x in pitchtools.list_numbered_chromatic_pitch_classes_in_expr(chord):
...     x
...
NumberedChromaticPitchClass(1)
NumberedChromaticPitchClass(2)
NumberedChromaticPitchClass(3)
```

Works with notes, chords, defective chords.

Return tuple or zero or more numbered chromatic pitch-classes.

21.3.94 pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range

pitchtools.**list_octave_transpositions_of_pitch_carrier_within_pitch_range**(*pitch_carrier*, *pitch_range*)

New in version 1.1. List octave transpositions of *pitch_carrier* in *pitch_range*:

```
>>> chord = Chord("<c' d' e'>4")
>>> pitch_range = pitchtools.PitchRange(0, 48)
```

```
>>> result = pitchtools.list_octave_transpositions_of_pitch_carrier_within_pitch_range(
...     chord, pitch_range)
```

```
>>> for chord in result:
...     chord
...
Chord("<c' d' e'>4")
Chord("<c'' d'' e''>4")
Chord("<c''' d''' e'''>4")
Chord("<c'''' d'''' e''''>4")
```

Return list of newly created *pitch_carrier* objects.

21.3.95 pitchtools.list_ordered_named_chromatic_pitch_pairs_from_expr_1_to_expr_2

pitchtools.**list_ordered_named_chromatic_pitch_pairs_from_expr_1_to_expr_2**(*expr_1*, *expr_2*)

New in version 2.0. List ordered named chromatic pitch pairs from *expr_1* to *expr_2*:

```
>>> chord_1 = Chord([0, 1, 2], (1, 4))
>>> chord_2 = Chord([3, 4], (1, 4))
```

```
>>> for pair in pitchtools.list_ordered_named_chromatic_pitch_pairs_from_expr_1_to_expr_2(
...     chord_1, chord_2):
...     pair
(NamedChromaticPitch("c'"), NamedChromaticPitch("ef'"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("e'"))
(NamedChromaticPitch("cs'"), NamedChromaticPitch("ef'"))
(NamedChromaticPitch("cs'"), NamedChromaticPitch("e'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("ef'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("e'"))
```

Return generator.

21.3.96 pitchtools.list_unordered_named_chromatic_pitch_pairs_in_expr

pitchtools.**list_unordered_named_chromatic_pitch_pairs_in_expr**(*expr*)

New in version 2.0. List unordered named chromatic pitch pairs in *expr*:

```
>>> chord = Chord("<c' cs' d' ef'>4")
```

```
>>> for pair in pitchtools.list_unordered_named_chromatic_pitch_pairs_in_expr(chord):
...     pair
...
(NamedChromaticPitch("c'"), NamedChromaticPitch("cs'"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("d'"))
(NamedChromaticPitch("c'"), NamedChromaticPitch("ef'"))
(NamedChromaticPitch("cs'"), NamedChromaticPitch("d'"))
(NamedChromaticPitch("cs'"), NamedChromaticPitch("ef'"))
(NamedChromaticPitch("d'"), NamedChromaticPitch("ef'"))
```

Return generator.

21.3.97 pitchtools.make_n_middle_c_centered_pitches

pitchtools.**make_n_middle_c_centered_pitches**(*n*)

New in version 2.0. Make *n* middle-c centered pitches, where $0 < n$:

```
>>> for p in pitchtools.make_n_middle_c_centered_pitches(5): p
NamedChromaticPitch('f')
NamedChromaticPitch('a')
NamedChromaticPitch("c'")
NamedChromaticPitch("e'")
NamedChromaticPitch("g'")
```

```
>>> for p in pitchtools.make_n_middle_c_centered_pitches(4): p
NamedChromaticPitch('g')
NamedChromaticPitch('b')
NamedChromaticPitch("d'")
NamedChromaticPitch("f'")
```

Return list of zero or more named chromatic pitches.

21.3.98 pitchtools.named_chromatic_pitch_and_clef_to_staff_position_number

pitchtools.**named_chromatic_pitch_and_clef_to_staff_position_number**(*pitch*, *clef*)

New in version 2.0. Change named chromatic *pitch* and *clef* to staff position number:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> clef = contexttools.ClefMark('treble')
>>> for note in staff:
...   written_pitch = note.written_pitch
```

```
...     number = pitchtools.named_chromatic_pitch_and_clef_to_staff_position_number(
...         written_pitch, clef)
...     print '%s\t%s' % (written_pitch, number)
c'      -6
d'      -5
e'      -4
f'      -3
g'      -2
a'      -1
b'      0
c''     1
```

Return integer.

21.3.99 pitchtools.octave_number_to_octave_tick_string

pitchtools.**octave_number_to_octave_tick_string**(*octave_number*)

New in version 2.0. Change *octave_number* to octave tick string:

```
>>> for octave_number in range(-1, 9):
...     tick_string = pitchtools.octave_number_to_octave_tick_string(octave_number)
...     print "%s\t%s" % (octave_number, tick_string)
...
-1 ,,,,
0  ,,,
1  ,,
2  ,
3
4  '
5  ''
6  '''
7  ''''
8  '''''
```

Raise type error on noninteger input.

Return string.

21.3.100 pitchtools.octave_tick_string_to_octave_number

pitchtools.**octave_tick_string_to_octave_number**(*tick_string*)

New in version 2.0. Change *tick_string* to octave number:

```
>>> pitchtools.octave_tick_string_to_octave_number("'")
4
```

Raise type error on nonstring input.

Raise value error on input not of tick string format.

Return integer.

21.3.101 pitchtools.pentatonic_pitch_number_to_chromatic_pitch_number

pitchtools.**pentatonic_pitch_number_to_chromatic_pitch_number**(*pentatonic_scale_degree*, *transpose=1*, *phase=0*)

New in version 1.1. Changed *pentatonic_scale_degree* number to chromatic pitch number:

```
>>> for pentatonic_scale_degree in range(9):
...     result = pitchtools.pentatonic_pitch_number_to_chromatic_pitch_number(
...         pentatonic_scale_degree)
...     print '%s %4s' % (pentatonic_scale_degree, result)
...
0   1
```

```
1  3
2  6
3  8
4  10
5  13
6  15
7  18
8  20
```

Pentatonic scale degrees may be negative:

```
>>> for pentatonic_scale_degree in range(-1, -9, -1):
...     result = pitchtools.pentatonic_pitch_number_to_chromatic_pitch_number(
...         pentatonic_scale_degree)
...     print '%s %4s' % (pentatonic_scale_degree, result)
...
-1 -2
-2 -4
-3 -6
-4 -9
-5 -11
-6 -14
-7 -16
-8 -18
```

Return integer.

21.3.102 pitchtools.permute_named_chromatic_pitch_carrier_list_by_twelve_tone_row

pitchtools.**permute_named_chromatic_pitch_carrier_list_by_twelve_tone_row**(*pitches*, *row*)

New in version 2.0. Permute named chromatic pitch carrier list by twelve-tone *row*:

```
>>> notes = notetools.make_notes([17, -10, -2, 11], [Duration(1, 4)])
>>> row = pitchtools.TwelveToneRow([10, 0, 2, 6, 8, 7, 5, 3, 1, 9, 4, 11])
>>> pitchtools.permute_named_chromatic_pitch_carrier_list_by_twelve_tone_row(notes, row)
[Note('bf4'), Note('d4'), Note("f''4"), Note("b'4")]
```

Function works by reference only. No objects are copied.

Return list.

21.3.103 pitchtools.pitch_class_octave_number_string_to_chromatic_pitch_name

pitchtools.**pitch_class_octave_number_string_to_chromatic_pitch_name**(*pitch_class_octave_number_*

New in version 2.5. Change *pitch_class_octave_number_string* to chromatic pitch name:

```
>>> pitchtools.pitch_class_octave_number_string_to_chromatic_pitch_name('C#+2')
'ctqs,'
```

Return string.

21.3.104 pitchtools.register_chromatic_pitch_class_numbers_by_chromatic_pitch_number_

pitchtools.**register_chromatic_pitch_class_numbers_by_chromatic_pitch_number_aggregate**(*pi ag gr ga*

New in version 1.1. Register chromatic *pitch_class_numbers* by chromatic pitch-number *aggregate*:

```
>>> pitchtools.register_chromatic_pitch_class_numbers_by_chromatic_pitch_number_aggregate(
...     [10, 0, 2, 6, 8, 7, 5, 3, 1, 9, 4, 11],
...     [10, 19, 20, 23, 24, 26, 27, 29, 30, 33, 37, 40])
[10, 24, 26, 30, 20, 19, 29, 27, 37, 33, 40, 23]
```

Return list of zero or more chromatic pitch numbers.

21.3.105 pitchtools.respell_named_chromatic_pitches_in_expr_with_flats

pitchtools.**respell_named_chromatic_pitches_in_expr_with_flats**(*expr*)

New in version 1.1. Respell named chromatic pitches in *expr* with flats:

```
>>> staff = Staff(notetools.make_repeated_notes(6))
>>> pitchtools.set_ascending_named_chromatic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    c'8
    cs'8
    d'8
    ef'8
    e'8
    f'8
}
```

```
>>> pitchtools.respell_named_chromatic_pitches_in_expr_with_flats(staff)
```

```
>>> f(staff)
\new Staff {
    c'8
    df'8
    d'8
    ef'8
    e'8
    f'8
}
```

Return none.

21.3.106 pitchtools.respell_named_chromatic_pitches_in_expr_with_sharps

pitchtools.**respell_named_chromatic_pitches_in_expr_with_sharps**(*expr*)

New in version 1.1. Respell named chromatic pitches in *expr* with sharps:

```
>>> staff = Staff(notetools.make_repeated_notes(6))
>>> pitchtools.set_ascending_named_chromatic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    c'8
    cs'8
    d'8
    ef'8
    e'8
    f'8
}
```

```
>>> pitchtools.respell_named_chromatic_pitches_in_expr_with_sharps(staff)
```

```
>>> f(staff)
\new Staff {
    c'8
    cs'8
    d'8
    ds'8
    e'8
    f'8
}
```

Return none.

21.3.107 pitchtools.set_ascending_named_chromatic_pitches_on_tie_chains_in_expr

pitchtools.**set_ascending_named_chromatic_pitches_on_tie_chains_in_expr**(*expr*)
New in version 1.1. Set ascending named chromatic pitches on nontied pitched components in *expr*:

```
>>> voice = Voice(notetools.make_notes(0, [(5, 32)] * 4))
>>> pitchtools.set_ascending_named_chromatic_pitches_on_tie_chains_in_expr(voice)
```

```
>>> f(voice)
\new Voice {
    c'8 ~
    c'32
    cs'8 ~
    cs'32
    d'8 ~
    d'32
    ef'8 ~
    ef'32
}
```

Used primarily in generating test file examples.

Return none.

21.3.108 pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr

pitchtools.**set_ascending_named_diatonic_pitches_on_tie_chains_in_expr**(*expr*, *key_signature=None*)
New in version 1.1. Set ascending named diatonic pitches on nontied pitched components in *expr*:

```
>>> staff = Staff(notetools.make_notes(0, [(5, 32)] * 4))
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'32
    d'8 ~
    d'32
    e'8 ~
    e'32
    f'8 ~
    f'32
}
```

Used primarily in generating test file examples. New in version 2.0: Optional *key_signature* keyword argument. Return none.

21.3.109 pitchtools.set_default_accidental_spelling

pitchtools.**set_default_accidental_spelling**(*spelling='mixed'*)
New in version 1.1. Set default accidental spelling to sharps:

```
>>> pitchtools.set_default_accidental_spelling('sharps')
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("cs''4"), Note("ds''4")]
```

Set default accidental spelling to flats:

```
>>> pitchtools.set_default_accidental_spelling('flats')
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("df''4"), Note("ef''4")]
```

Set default accidental spelling to mixed:

```
>>> pitchtools.set_default_accidental_spelling()
```

```
>>> [Note(13, (1, 4)), Note(15, (1, 4))]
[Note("cs''4"), Note("ef''4")]
```

Mixed is system default.

Mixed test case must appear last here for doc tests to check correctly.

Return none.

21.3.110 pitchtools.set_written_pitch_of_pitched_components_in_expr

pitchtools.**set_written_pitch_of_pitched_components_in_expr**(*expr*, *written_pitch=0*)

New in version 2.9. Set written pitch of pitched components in *expr* to *written_pitch*:

```
>>> staff = Staff("c' d' e' f'")
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

```
>>> pitchtools.set_written_pitch_of_pitched_components_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    c'4
    c'4
    c'4
    c'4
}
```

Use as a way of neutralizing pitch information in an arbitrary piece of score.

Return none.

21.3.111 pitchtools.sort_named_chromatic_pitch_carriers_in_expr

pitchtools.**sort_named_chromatic_pitch_carriers_in_expr**(*pitch_carriers*)

New in version 2.0. List named chromatic pitch carriers in *expr* sorted by numbered chromatic pitch-class:

```
>>> chord = Chord([9, 11, 12, 14, 16], (1, 4))
>>> notes = chordtools.arpeggiate_chord(chord)
```

```
>>> pitchtools.sort_named_chromatic_pitch_carriers_in_expr(notes)
[Note("c''4"), Note("d''4"), Note("e''4"), Note("a'4"), Note("b'4")]
```

The elements in *pitch_carriers* are not changed in any way.

Return list.

21.3.112 pitchtools.spell_chromatic_interval_number

pitchtools.**spell_chromatic_interval_number**(*diatonic_interval_number*, *chromatic_interval_number*)

New in version 2.0. Spell *chromatic_interval_number* according to *diatonic_interval_number*:

```
>>> pitchtools.spell_chromatic_interval_number(2, 1)
MelodicDiatonicInterval('+m2')
```

Return melodic diatonic interval.

21.3.113 pitchtools.spell_chromatic_pitch_number

`pitchtools.`**`spell_chromatic_pitch_number`**(*chromatic_pitch_number*, *diatonic_pitch_class_name*)

New in version 1.1. Spell *chromatic_pitch_number* according to *diatonic_pitch_class_name*:

```
>>> pitchtools.spell_chromatic_pitch_number(14, 'c')
(Accidental('ss'), 5)
```

Return accidental / octave-number pair.

21.3.114 pitchtools.split_chromatic_pitch_class_name

`pitchtools.`**`split_chromatic_pitch_class_name`**(*chromatic_pitch_class_name*)

New in version 1.1. Change *chromatic_pitch_class_name* to diatonic pitch-class name / alphabetic accidental abbreviation pair:

```
>>> pitchtools.split_chromatic_pitch_class_name('cs')
('c', 's')
```

Return pair of strings.

21.3.115 pitchtools.suggest_clef_for_named_chromatic_pitches

`pitchtools.`**`suggest_clef_for_named_chromatic_pitches`**(*pitches*)

New in version 1.1. Suggest clef for named chromatic *pitches*:

```
>>> staff = Staff(notetools.make_notes(range(-12, -6), [(1, 4)]))
>>> pitchtools.suggest_clef_for_named_chromatic_pitches(staff)
ClefMark('bass')
```

Suggest clef based on minimal number of ledger lines.

Return clef mark.

21.3.116 pitchtools.symbolic_accidental_string_to_alphabetic_accidental_abbreviation

`pitchtools.`**`symbolic_accidental_string_to_alphabetic_accidental_abbreviation`**(*symbolic_accider*

New in version 2.5. Change *symbolic_accidental_string* to alphabetic accidental abbreviation:

```
>>> pitchtools.symbolic_accidental_string_to_alphabetic_accidental_abbreviation('#+')
'tqs'
```

None when *symbolic_accidental_string* is not a valid symbolic accidental string.

Return string or none.

21.3.117 pitchtools.transpose_chromatic_pitch_by_melodic_chromatic_interval_segment

`pitchtools.`**`transpose_chromatic_pitch_by_melodic_chromatic_interval_segment`**(*pitch*, *segment*)

New in version 2.0. Transpose chromatic *pitch* by melodic chromatic interval *segment*:

```
>>> ncp = pitchtools.NumberedChromaticPitch(0)
>>> mcis = pitchtools.MelodicChromaticIntervalSegment([0, -1, 2])
>>> pitchtools.transpose_chromatic_pitch_by_melodic_chromatic_interval_segment(ncp, mcis)
[NumberedChromaticPitch(0), NumberedChromaticPitch(-1), NumberedChromaticPitch(1)]
```

Transpose by each interval in *segment* such that each tranposition transposes the resulting pitch of the previous transposition.

Return list of numbered chromatic pitches.

21.3.118 pitchtools.transpose_chromatic_pitch_class_number_to_chromatic_pitch_number_

pitchtools.**transpose_chromatic_pitch_class_number_to_chromatic_pitch_number_neighbor**(*chro chro mat*

New in version 1.1. Transpose *chromatic_pitch_class_number* by octaves to nearest neighbor of *chromatic_pitch_number*:

```
>>> pitchtools.transpose_chromatic_pitch_class_number_to_chromatic_pitch_number_neighbor(
...     12, 4)
16
```

Resulting chromatic pitch number must be within one tritone of *chromatic_pitch_number*.

Return chromatic pitch number.

21.3.119 pitchtools.transpose_chromatic_pitch_number_by_octave_transposition_mapping

pitchtools.**transpose_chromatic_pitch_number_by_octave_transposition_mapping**(*chromatic_pitch_ map-ping*)

New in version 1.1. Transpose *chromatic_pitch_number* by the some number of octaves up or down. Derive correct number of octaves from *mapping* where *mapping* is a list of `(range_spec, octave)` pairs and `range_spec` is, in turn, a `(start, stop)` pair suitable to pass to the built-in Python `range()` function:

```
>>> mapping = [((-39, -13), 0), ((-12, 23), 12), ((24, 48), 24)]
```

The mapping given here comprises three `(range_spec, octave)` pairs. The first such pair is `((-39, -13), 0)` and can be read as follows: “any pitches between `-39` and `-13` should be transposed into the octave rooted at pitch `0`.” The octave rooted at pitch `0` equals the twelve pitches `range(0, 0 + 12)` or `[0, 1, ..., 10, 11]`.

The second `(range_spec, octave)` pair is `((-12, 23), 12)` and can be read as “any pitches between `-12` and `23` should be transposed into the octave rooted at pitch `12`,” with the octave rooted at pitch `12` equal to the twelve pitches `range(12, 12 + 12)` or `[12, 13, ..., 22, 23]`.

The third and last `(range_spec, octave)` pair is `((24, 48), 24)` and can be read as “any pitches between `24` and `48` should be transposed to the octave rooted at `24`,” with the octave rooted at `24` equal to the twelve pitches `range(24, 24, + 12)` or `[24, 25, ..., 34, 35]`.

The mapping given here divides the compass of the piano, from `-39` to `48`, into three disjunct subranges and then explains how to transpose pitches found in any of those three disjunct subranges. This means that, for example, all the f-sharps within the range of the piano now undergo a known transposition under *mapping* as defined here:

```
>>> pitchtools.transpose_chromatic_pitch_number_by_octave_transposition_mapping(
...     -30, mapping)
6
```

We verify that pitch `-30` should map to pitch `6` by noticing that pitch `-30` falls in the first of the three subranges defined by *mapping* from `-39` to `-13` and then noting that *mapping* sends pitches with that subrange to the octave rooted at pitch `0`. The octave transposition of `-30` that falls within the octave rooted at `0` is `6`:

```
>>> pitchtools.transpose_chromatic_pitch_number_by_octave_transposition_mapping(
...     -18, mapping)
6
```

Likewise, *mapping* sends pitch `-18` to pitch `6` because pitch `-18` falls in the same subrange from `-39` to `-13` as did pitch `-39` and so undergoes the same transposition to the octave rooted at `0`.

In this way we can map all f-sharps from `-39` to `48` according to *mapping*:

```
>>> pitch_numbers = [-30, -18, -6, 6, 18, 30, 42]
>>> for n in pitch_numbers:
...   n, pitchtools.transpose_chromatic_pitch_number_by_octave_transposition_mapping(
...         n, mapping)
...
(-30, 6)
(-18, 6)
(-6, 18)
(6, 18)
(18, 18)
(30, 30)
(42, 30)
```

And so on.

Return chromatic pitch number.

21.3.120 pitchtools.transpose_named_chromatic_pitch_by_melodic_chromatic_interval_and

pitchtools.**transpose_named_chromatic_pitch_by_melodic_chromatic_interval_and_respell**(*pitc staff mel*

New in version 1.1. Transpose named chromatic pitch by *melodic_chromatic_interval* and respell *staff_spaces* above or below:

```
>>> pitch = pitchtools.NamedChromaticPitch(0)
```

```
>>> pitchtools.transpose_named_chromatic_pitch_by_melodic_chromatic_interval_and_respell(
...     pitch, 1, 0.5)
NamedChromaticPitch("dtqf'")
```

Return new named chromatic pitch.

21.3.121 pitchtools.transpose_pitch_carrier_by_melodic_interval

pitchtools.**transpose_pitch_carrier_by_melodic_interval**(*pitch_carrier*, *melodic_interval*)

New in version 2.0. Transpose *pitch_carrier* by diatonic *melodic_interval*:

```
>>> chord = Chord("<c' e' g'>4")
```

```
>>> pitchtools.transpose_pitch_carrier_by_melodic_interval(chord, '+m2')
Chord("<df' f' af'>4")
```

Transpose *pitch_carrier* by chromatic *melodic_interval*:

```
>>> chord = Chord("<c' e' g'>4")
```

```
>>> pitchtools.transpose_pitch_carrier_by_melodic_interval(chord, 1)
Chord("<cs' f' af'>4")
```

Return non-pitch-carrying input unchaged:

```
>>> rest = Rest('r4')
```

```
>>> pitchtools.transpose_pitch_carrier_by_melodic_interval(rest, 1)
Rest('r4')
```

Return *pitch_carrier*.

21.3.122 pitchtools.transpose_pitch_expr_into_pitch_range

pitchtools.**transpose_pitch_expr_into_pitch_range**(*pitch_expr*, *pitch_range*)

New in version 2.0. Transpose *pitch_expr* into *pitch_range*:

```
>>> pitchtools.transpose_pitch_expr_into_pitch_range(
...     [-2, -1, 13, 14], pitchtools.PitchRange(0, 12))
[10, 11, 1, 2]
```

Return new *pitch_expr* object.

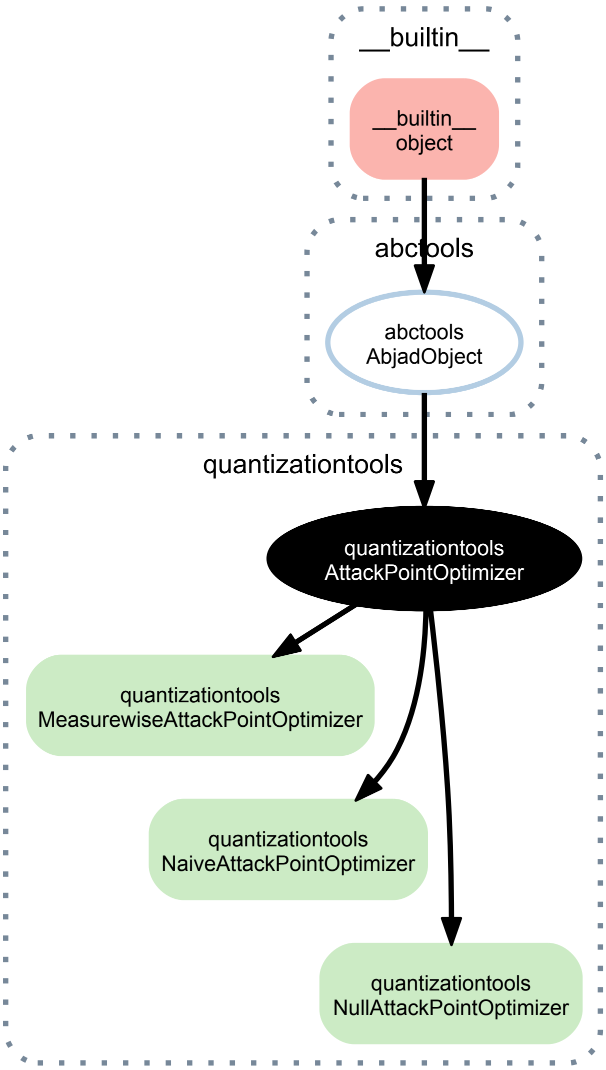
CHAPTER

TWENTYTWO

QUANTIZATIONTOOLS

22.1 Abstract Classes

22.1.1 quantizationtools.AttackPointOptimizer



class `quantizationtools.`**`AttackPointOptimizer`**

Abstract attack-point optimizer class from which concrete attack-point optimizer classes inherit.

Attack-point optimizers may alter the number, order, and individual durations of leaves in a tie chain, but may not alter the overall duration of that tie chain.

They effectively "clean up" notation, post-quantization.

Read-only properties

`AttackPointOptimizer.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`AttackPointOptimizer.`**`__call__`**`(`*expr*`)`

`AttackPointOptimizer.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`AttackPointOptimizer.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AttackPointOptimizer.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`AttackPointOptimizer.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AttackPointOptimizer.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

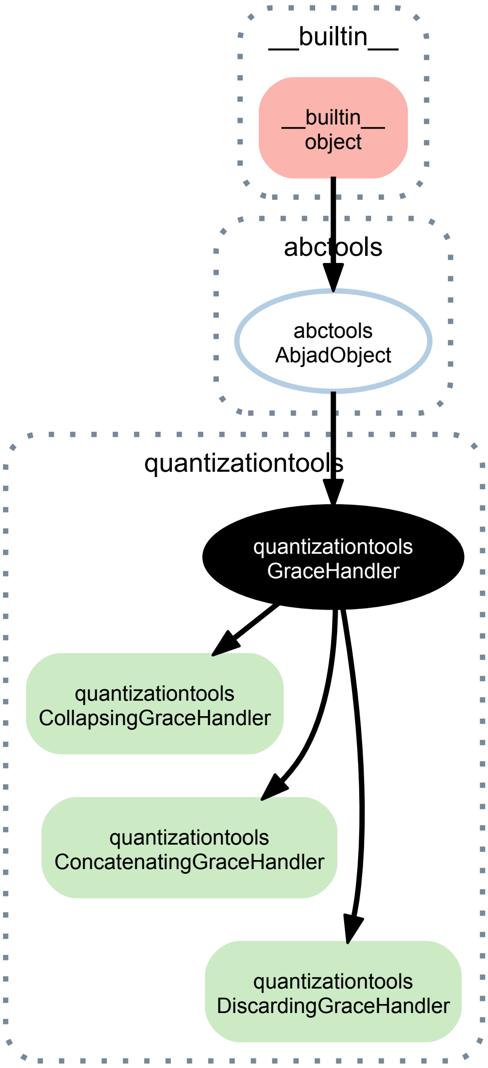
`AttackPointOptimizer.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`AttackPointOptimizer.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

22.1.2 quantizationtools.GraceHandler



class `quantizationtools.`**`GraceHandler`**

Abstract base class from which concrete `GraceHandler` subclasses inherit.

Determines what pitch, if any, will be selected from a list of `QEvents` to be applied to an attack-point generated by a `QGrid`, and whether there should be a `GraceContainer` attached to that attack-point.

When called on a sequence of `QEvents`, `GraceHandler` subclasses should return a pair, where the first item of the pair is a sequence of pitch tokens or `None`, and where the second item of the pair is a `GraceContainer` instance or None.

Read-only properties

`GraceHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`GraceHandler.`**`__call__`**(*q_events*)

`GraceHandler.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`GraceHandler.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`GraceHandler.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`GraceHandler.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`GraceHandler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

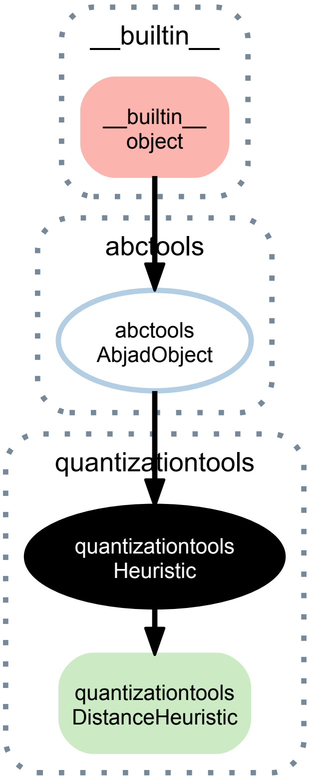
`GraceHandler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`GraceHandler.__repr__()`
Interpreter representation of Abjad object.

Return string.

22.1.3 quantizationtools.Heuristic



class quantizationtools.**Heuristic**

Abstract base class from which concrete Heuristic subclases inherit.

Heuristics rank QGrids according to the criteria they encapsulate.

They provide the means by which the quantizer selects a single QGrid from all computed QGrids for any given QTargetBeat to represent that beat.

Read-only properties

Heuristic.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

Heuristic.**__call__**(*q_target_beats*)

Heuristic.**__eq__**(*expr*)

True when id(self) equals id(expr).

Return boolean.

Heuristic.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Heuristic.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

Heuristic.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Heuristic.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Heuristic.**__ne__**(*expr*)

Defined equal to the opposite of equality.

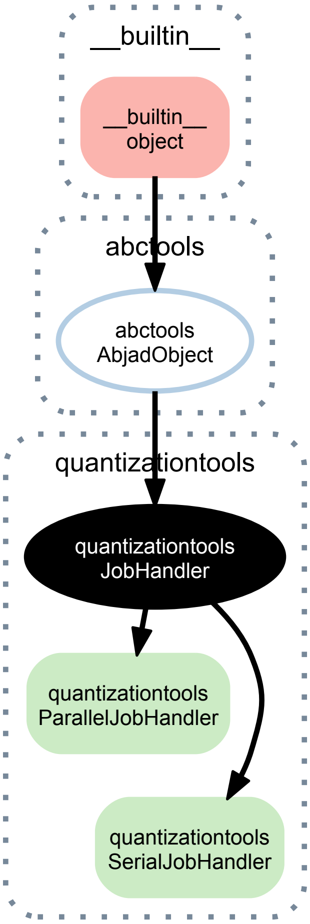
Return boolean.

Heuristic.**__repr__**()

Interpreter representation of Abjad object.

Return string.

22.1.4 quantizationtools.JobHandler



class `quantizationtools.`**`JobHandler`**

Abstact job handler class from which concrete job handlers inherit.

`JobHandlers` control how `QuantizationJob` instances are processed by the `Quantizer`, either serially or in parallel.

Read-only properties

`JobHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`JobHandler.`**`__call__`**(*jobs*)

`JobHandler.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`JobHandler.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`JobHandler.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`JobHandler.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`JobHandler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

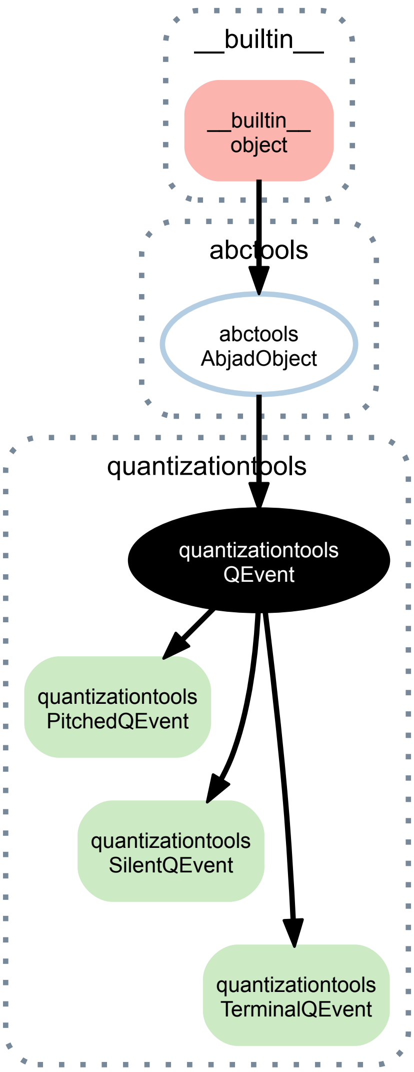
`JobHandler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`JobHandler.__repr__()`
Interpreter representation of Abjad object.

Return string.

22.1.5 quantizationtools.QEvent



class `quantizationtools.`**`QEvent`**`(`*offset*, *index=None*`)`

Abstract base class from which concrete `QEvent` subclasses inherit.

Represents an attack point to be quantized.

All `QEvents` possess a rational offset in milliseconds, and an optional index for disambiguating events which fall on the same offset in a `QGrid`.

Read-only properties

`QEvent.`**`index`**

The optional index, for sorting QEvents with identical offsets.

`QEvent.`**`offset`**

The offset in milliseconds of the event.

`QEvent.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`QEvent.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`QEvent.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`QEvent.`**`__getstate__`**`()`

`QEvent.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`QEvent.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`QEvent.`**`__lt__`**`(`*expr*`)`

`QEvent.`**`__ne__`**`(`*expr*`)`

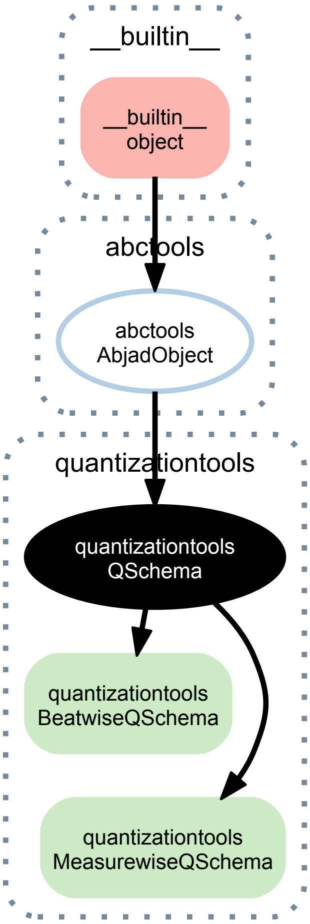
Defined equal to the opposite of equality.

Return boolean.

`QEvent.`**`__repr__`**`()`

`QEvent.`**`__setstate__`**`(`*state*`)`

22.1.6 quantizationtools.QSchema



class `quantizationtools.`**`QSchema`**(**args*, ***kwargs*)

The *schema* for a quantization run.

`QSchema` allows for the specification of quantization settings diachronically, at any time-step of the quantization process.

In practice, this provides a means for the composer to change the tempo, search-tree, time-signature etc., effectively creating a template into which quantized rhythms can be "poured", without yet knowing what those rhythms might be, or even how much time the ultimate result will take. Like Abjad's `ContextMarks`, the settings made at any given time-step via a `QSchema` instance are understood to persist until changed.

All concrete `QSchema` subclasses strongly implement default values for all of their parameters.

QSchema is abstract.

Read-only properties

`QSchema.`**`item_klass`**

The schema's item class.

`QSchema.`**`items`**

The item dictionary.

`QSchema.`**`search_tree`**

The default search tree.

`QSchema.`**`storage_format`**

Storage format of Abjad object.

Return string.

QSchema.**target_item_klass**
The schema's target class' item class.

QSchema.**target_klass**
The schema's target class.

QSchema.**tempo**
The default tempo.

Special methods

QSchema.**__call__**(*duration*)

QSchema.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

QSchema.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QSchema.**__getitem__**(*i*)

QSchema.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

QSchema.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QSchema.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

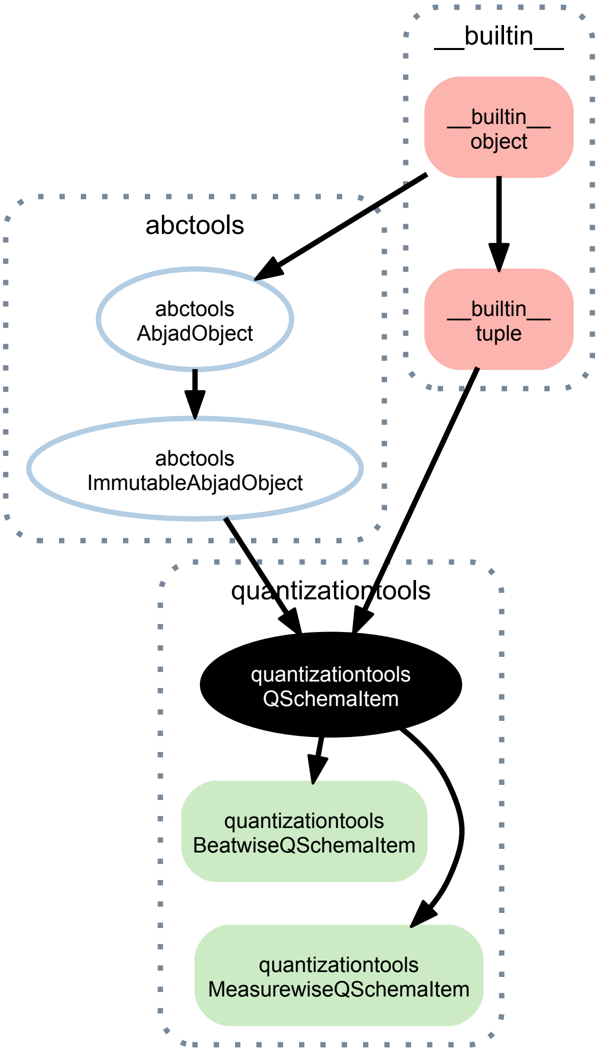
Raise exception.

QSchema.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

QSchema.**__repr__**()

22.1.7 quantizationtools.QSchemaItem



class `quantizationtools.`**`QSchemaItem`**(**args*, ***kwargs*)

QSchemaItem represents a change of state in the timeline of a quantization process.

QSchemaItem is abstract and immutable.

Read-only properties

`QSchemaItem.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`QSchemaItem.`**`count`**(*value*) → integer – return number of occurrences of value

`QSchemaItem.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.

Raises ValueError if the value is not present.

Special methods

`QSchemaItem.__add__()`
x.__add__(y) <==> x+y

`QSchemaItem.__contains__()`
x.__contains__(y) <==> y in x

`QSchemaItem.__eq__()`
x.__eq__(y) <==> x==y

`QSchemaItem.__ge__()`
x.__ge__(y) <==> x>=y

`QSchemaItem.__getitem__()`
x.__getitem__(y) <==> x[y]

`QSchemaItem.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`QSchemaItem.__gt__()`
x.__gt__(y) <==> x>y

`QSchemaItem.__hash__()` <==> *hash(x)*

`QSchemaItem.__iter__()` <==> *iter(x)*

`QSchemaItem.__le__()`
x.__le__(y) <==> x<=y

`QSchemaItem.__len__()` <==> *len(x)*

`QSchemaItem.__lt__()`
x.__lt__(y) <==> x<y

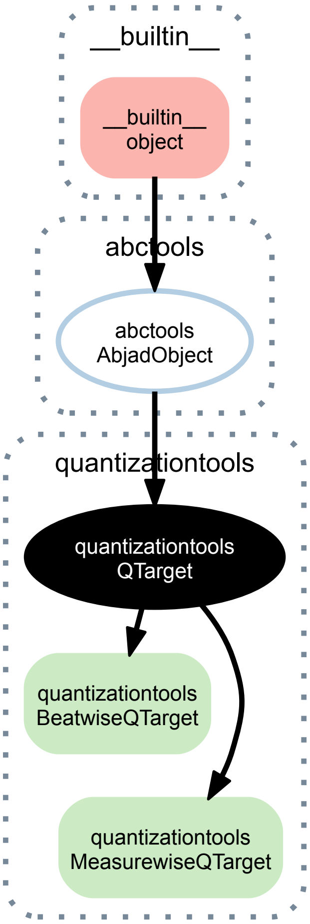
`QSchemaItem.__mul__()`
x.__mul__(n) <==> x*n

`QSchemaItem.__ne__()`
x.__ne__(y) <==> x!=y

`QSchemaItem.__repr__()`

`QSchemaItem.__rmul__()`
x.__rmul__(n) <==> n*x

22.1.8 quantizationtools.QTarget



class `quantizationtools.`**`QTarget`**(*items*)

Abstract base class from which concrete `QTarget` subclasses inherit.

`QTarget` is created by a concrete `QSchema` instance, and represents the mold into which the timepoints contained by a `QSequence` instance will be poured, as structured by that `QSchema` instance.

Not composer-safe.

Used internally by the `Quantizer`.

Read-only properties

`QTarget.`**`beats`**

`QTarget.`**`duration_in_ms`**

`QTarget.`**`item_klass`**

`QTarget.`**`items`**

`QTarget.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`QTarget.`**`__call__`**(*q_event_sequence*, *grace_handler=None*, *heuristic=None*, *job_handler=None*, *attack_point_optimizer=None*, *attach_tempo_marks=True*)

QTarget.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

QTarget.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QTarget.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

QTarget.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QTarget.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

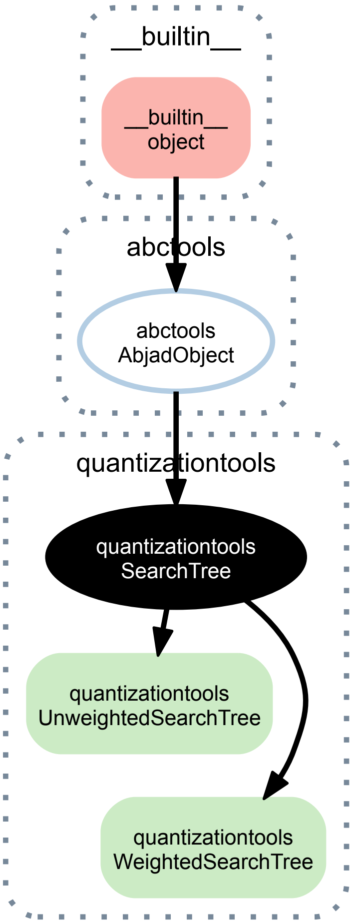
QTarget.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

QTarget.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.1.9 quantizationtools.SearchTree



class `quantizationtools.`**`SearchTree`**(*definition=None*)

Abstract base class from which concrete `SearchTree` subclasses inherit.

`SearchTrees` encapsulate strategies for generating collections of `QGrids`, given a set of `QEventProxy` instances as input.

They allow composers to define the degree and quality of nested rhythmic subdivisions in the quantization output. That is to say, they allow composers to specify what sorts of tuplets and ratios of pulses may be contained within other tuplets, to arbitrary levels of nesting.

Read-only properties

`SearchTree.`**`default_definition`**

The default search tree definition.

Return dictionary.

`SearchTree.`**`definition`**

The search tree definition.

Return dictionary.

`SearchTree.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

SearchTree.**__call__**(*q_grid*)

SearchTree.**__eq__**(*expr*)

SearchTree.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SearchTree.**__getstate__**()

SearchTree.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SearchTree.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SearchTree.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SearchTree.**__ne__**(*expr*)
Defined equal to the opposite of equality.

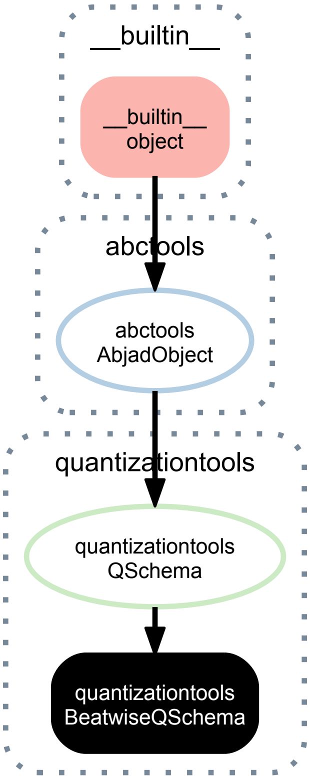
Return boolean.

SearchTree.**__repr__**()

SearchTree.**__setstate__**(*state*)

22.2 Concrete Classes

22.2.1 quantizationtools.BeatwiseQSchema



class `quantizationtools.`**`BeatwiseQSchema`**(**args*, ***kwargs*)

Concrete `QSchema` subclass which treats "beats" as its time-step unit:

```
>>> q_schema = quantizationtools.BeatwiseQSchema()
```

Without arguments, it uses smart defaults:

```
>>> q_schema
quantizationtools.BeatwiseQSchema(
    beatspan=durationtools.Duration(1, 4),
    search_tree=quantizationtools.UnweightedSearchTree(
        definition={   2: {   2: {   2: {   2: None}, 3: None}, 3: None, 5: None, 7: None},
            3: {   2: {   2: None}, 3: None, 5: None},
            5: {   2: None, 3: None},
            7: {   2: None},
            11: None,
            13: None}
        ),
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
        ),
    )
```

Each time-step in a `BeatwiseQSchema` is composed of three settings:

- `beatspan`
- `search_tree`
- `tempo`

These settings can be applied as global defaults for the schema via keyword arguments, which persist until overridden:

```
>>> beatspan = Duration(5, 16)
>>> search_tree = quantizationtools.UnweightedSearchTree({7: None})
>>> tempo = contexttools.TempoMark((1, 4), 54)
>>> q_schema = quantizationtools.BeatwiseQSchema(
...     beatspan=beatspan,
...     search_tree=search_tree,
...     tempo=tempo,
...     )
```

The computed value at any non-negative time-step can be found by subscripting:

```
>>> index = 0
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
beatspan: 5/16
search_tree: UnweightedSearchTree(
    definition={   7: None}
    )
tempo: TempoMark(Duration(1, 4), 54)
```

```
>>> index = 1000
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
beatspan: 5/16
search_tree: UnweightedSearchTree(
    definition={   7: None}
    )
tempo: TempoMark(Duration(1, 4), 54)
```

Per-time-step settings can be applied in a variety of ways.

Instantiating the schema via `*args` with a series of either `BeatwiseQSchemaItem` instances, or dictionaries which could be used to instantiate `BeatwiseQSchemaItem` instances, will apply those settings sequentially, starting from time-step `0`:

```
>>> a = {'beatspan': Duration(5, 32)}
>>> b = {'beatspan': Duration(3, 16)}
>>> c = {'beatspan': Duration(1, 8)}
```

```
>>> q_schema = quantizationtools.BeatwiseQSchema(a, b, c)
```

```
>>> q_schema[0]['beatspan']
Duration(5, 32)
```

```
>>> q_schema[1]['beatspan']
Duration(3, 16)
```

```
>>> q_schema[2]['beatspan']
Duration(1, 8)
```

```
>>> q_schema[3]['beatspan']
Duration(1, 8)
```

Similarly, instantiating the schema from a single dictionary, consisting of integer:specification pairs, or a sequence via `*args` of (integer, specification) pairs, allows for applying settings to non-sequential time-steps:

```
>>> a = {'search_tree': quantizationtools.UnweightedSearchTree({2: None})}
>>> b = {'search_tree': quantizationtools.UnweightedSearchTree({3: None})}
```

```
>>> settings = {
...     2: a,
...     4: b,
... }
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(settings)
```

```
>>> q_schema[0]['search_tree']
UnweightedSearchTree(
    definition={   2: {   2: {   2: {   2: None}, 3: None}, 3: None, 5: None, 7: None},
        3: {   2: {   2: None}, 3: None, 5: None},
        5: {   2: None, 3: None},
        7: {   2: None},
        11: None,
        13: None}
    )
```

```
>>> q_schema[1]['search_tree']
UnweightedSearchTree(
    definition={   2: {   2: {   2: {   2: None}, 3: None}, 3: None, 5: None, 7: None},
        3: {   2: {   2: None}, 3: None, 5: None},
        5: {   2: None, 3: None},
        7: {   2: None},
        11: None,
        13: None}
    )
```

```
>>> q_schema[2]['search_tree']
UnweightedSearchTree(
    definition={   2: None}
    )
```

```
>>> q_schema[3]['search_tree']
UnweightedSearchTree(
    definition={   2: None}
    )
```

```
>>> q_schema[4]['search_tree']
UnweightedSearchTree(
    definition={   3: None}
    )
```

```
>>> q_schema[1000]['search_tree']
UnweightedSearchTree(
    definition={   3: None}
    )
```

The following is equivalent to the above schema definition:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     (2, {'search_tree': quantizationtools.UnweightedSearchTree({2: None})}),
...     (4, {'search_tree': quantizationtools.UnweightedSearchTree({3: None})}),
...     )
```

Return `BeatwiseQSchema` instance.

Read-only properties

`BeatwiseQSchema.`**`beatspan`**
The default beatspan.

`BeatwiseQSchema.`**`item_klass`**
The schema's item class.

`BeatwiseQSchema.`**`items`**
The item dictionary.

`BeatwiseQSchema.`**`search_tree`**
The default search tree.

`BeatwiseQSchema.`**`storage_format`**
Storage format of Abjad object.

Return string.

`BeatwiseQSchema.`**`target_item_klass`**

BeatwiseQSchema.**target_klass**

BeatwiseQSchema.**tempo**
The default tempo.

Special methods

BeatwiseQSchema.**__call__**(*duration*)

BeatwiseQSchema.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

BeatwiseQSchema.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BeatwiseQSchema.**__getitem__**(*i*)

BeatwiseQSchema.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BeatwiseQSchema.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BeatwiseQSchema.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

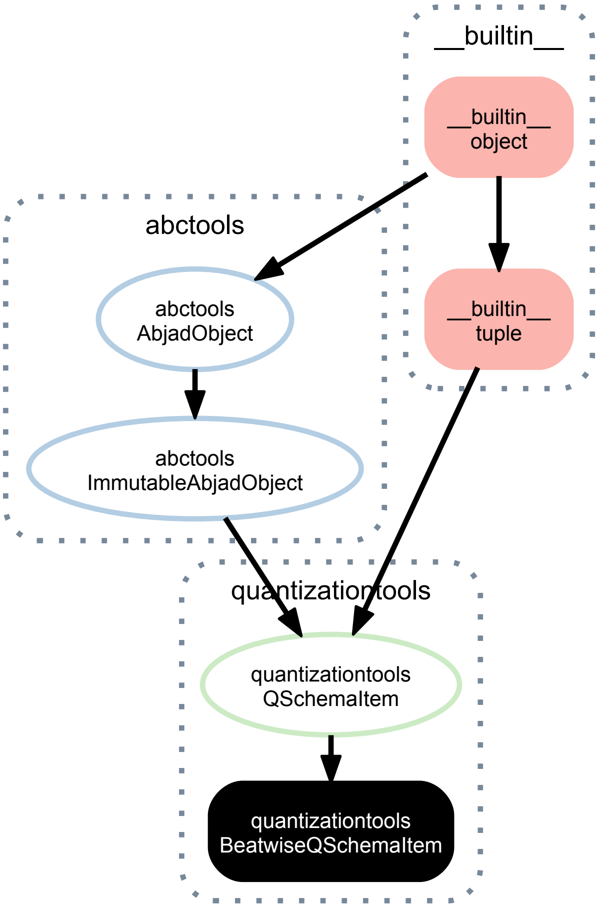
Raise exception.

BeatwiseQSchema.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

BeatwiseQSchema.**__repr__**()

22.2.2 quantizationtools.BeatwiseQSchemaItem



class `quantizationtools.BeatwiseQSchemaItem`(*beatspan=None*, *search_tree=None*, *tempo=None*)

BeatwiseQSchemaItem represents a change of state in the timeline of an unmetered quantization process.

```
>>> quantizationtools.BeatwiseQSchemaItem()
BeatwiseQSchemaItem()
```

Define a change in tempo:

```
>>> quantizationtools.BeatwiseQSchemaItem(tempo=((1, 4), 60))
BeatwiseQSchemaItem(
    tempo=contexttools.TempoMark(
            durationtools.Duration(1, 4),
            60
            ),
    )
```

Define a change in beatspan:

```
>>> quantizationtools.BeatwiseQSchemaItem(beatspan=(1, 8))
BeatwiseQSchemaItem(
    beatspan=durationtools.Duration(1, 8),
    )
```

BeatwiseQSchemaItem is immutable.

Return *BeatwiseQSchemaItem* instance.

Read-only properties

BeatwiseQSchemaItem.**beatspan**
The optionally defined beatspan duration.

Return *Duration* or *None*.

BeatwiseQSchemaItem.**search_tree**
The optionally defined search tree.

Return *OldSearchTree* or *None*.

BeatwiseQSchemaItem.**storage_format**
Storage format of Abjad object.

Return string.

BeatwiseQSchemaItem.**tempo**
The optionally defined *TempoMark*.

Return *TempoMark* or *None*.

Methods

BeatwiseQSchemaItem.**count** (*value*) → integer – return number of occurrences of value

BeatwiseQSchemaItem.**index** (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

BeatwiseQSchemaItem.**__add__** ()
x.__add__(y) <==> x+y

BeatwiseQSchemaItem.**__contains__** ()
x.__contains__(y) <==> y in x

BeatwiseQSchemaItem.**__eq__** ()
x.__eq__(y) <==> x==y

BeatwiseQSchemaItem.**__ge__** ()
x.__ge__(y) <==> x>=y

BeatwiseQSchemaItem.**__getitem__** ()
x.__getitem__(y) <==> x[y]

BeatwiseQSchemaItem.**__getslice__** ()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

BeatwiseQSchemaItem.**__gt__** ()
x.__gt__(y) <==> x>y

BeatwiseQSchemaItem.**__hash__** () <==> *hash(x)*

BeatwiseQSchemaItem.**__iter__** () <==> *iter(x)*

BeatwiseQSchemaItem.**__le__** ()
x.__le__(y) <==> x<=y

BeatwiseQSchemaItem.**__len__** () <==> *len(x)*

BeatwiseQSchemaItem.**__lt__** ()
x.__lt__(y) <==> x<y

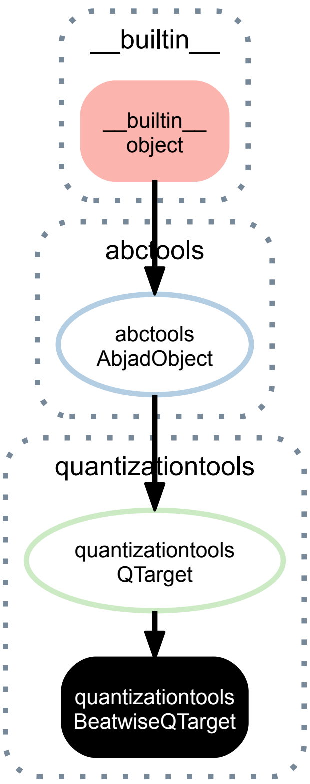
BeatwiseQSchemaItem.**__mul__**()
x.__mul__(n) <==> x*n

BeatwiseQSchemaItem.**__ne__**()
x.__ne__(y) <==> x!=y

BeatwiseQSchemaItem.**__repr__**()

BeatwiseQSchemaItem.**__rmul__**()
x.__rmul__(n) <==> n*x

22.2.3 quantizationtools.BeatwiseQTarget



class quantizationtools.**BeatwiseQTarget**(*items*)
A beat-wise quantization target.

Not composer-safe.

Used internally by `Quantizer`.

Return `BeatwiseQTarget` instance.

Read-only properties

BeatwiseQTarget.**beats**

BeatwiseQTarget.**duration_in_ms**

BeatwiseQTarget.**item_klass**

BeatwiseQTarget.**items**

BeatwiseQTarget.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

`BeatwiseQTarget.__call__`(*q_event_sequence*, *grace_handler=None*, *heuristic=None*, *job_handler=None*, *attack_point_optimizer=None*, *attach_tempo_marks=True*)

`BeatwiseQTarget.__eq__`(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`BeatwiseQTarget.__ge__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`BeatwiseQTarget.__gt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`BeatwiseQTarget.__le__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`BeatwiseQTarget.__lt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

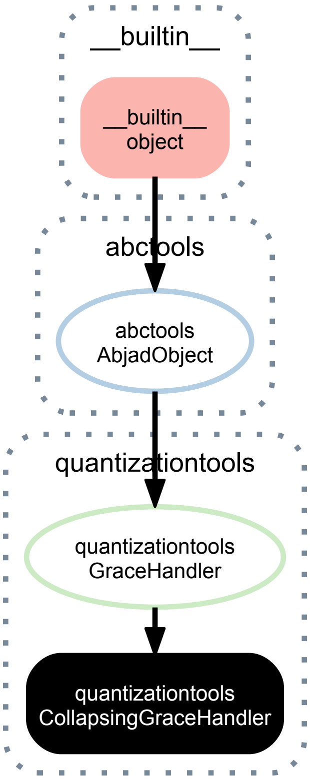
`BeatwiseQTarget.__ne__`(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`BeatwiseQTarget.__repr__`()
Interpreter representation of Abjad object.

Return string.

22.2.4 quantizationtools.CollapsingGraceHandler



class `quantizationtools.`**`CollapsingGraceHandler`**

A GraceHandler which collapses pitch information into a single Chord, rather than creating a GraceContainer.

Return `CollapsingGraceHandler` instance.

Read-only properties

`CollapsingGraceHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`CollapsingGraceHandler.`**`__call__`**(*q_events*)

`CollapsingGraceHandler.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`CollapsingGraceHandler.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`CollapsingGraceHandler.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`CollapsingGraceHandler.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`CollapsingGraceHandler.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`CollapsingGraceHandler.`**`__ne__`**`(`*expr*`)`

Defined equal to the opposite of equality.

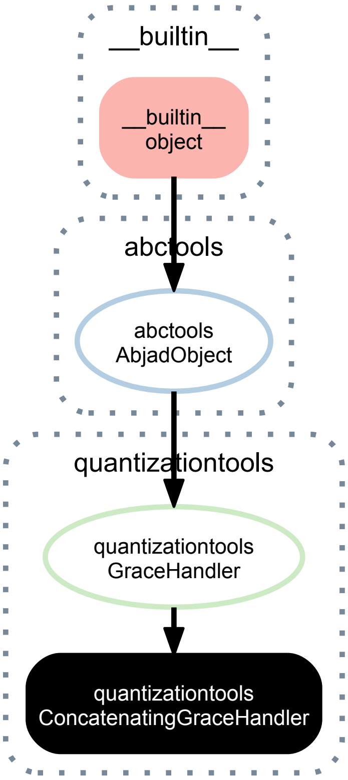
Return boolean.

`CollapsingGraceHandler.`**`__repr__`**`()`

Interpreter representation of Abjad object.

Return string.

22.2.5 quantizationtools.ConcatenatingGraceHandler



class `quantizationtools.`**`ConcatenatingGraceHandler`**`(`*grace_duration=None*`)`

Concrete `GraceHandler` subclass which concatenates all but the final `QEvent` attached to a `QGrid` offset into a `GraceContainer`, using a fixed leaf duration `duration`.

When called, it returns pitch information of final `QEvent`, and the generated `GraceContainer`, if any.

Return `ConcatenatingGraceHandler` instance.

Read-only properties

`ConcatenatingGraceHandler.`**`grace_duration`**

`ConcatenatingGraceHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

ConcatenatingGraceHandler.**__call__**(*q_events*)

ConcatenatingGraceHandler.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ConcatenatingGraceHandler.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ConcatenatingGraceHandler.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ConcatenatingGraceHandler.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ConcatenatingGraceHandler.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

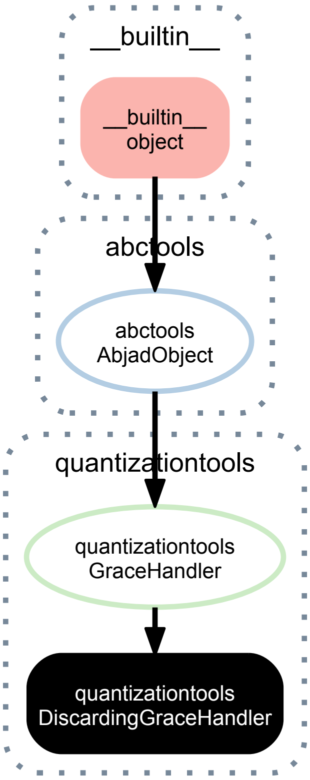
ConcatenatingGraceHandler.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ConcatenatingGraceHandler.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.6 quantizationtools.DiscardingGraceHandler



class `quantizationtools.`**`DiscardingGraceHandler`**

Concrete `GraceHandler` subclass which discards all but final `QEvent` attached to an offset.

Does not create `GraceContainers`.

Return `DiscardingGraceHandler` instance.

Read-only properties

`DiscardingGraceHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`DiscardingGraceHandler.`**`__call__`**`(`*q_events*`)`

`DiscardingGraceHandler.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`DiscardingGraceHandler.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DiscardingGraceHandler.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`DiscardingGraceHandler.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DiscardingGraceHandler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

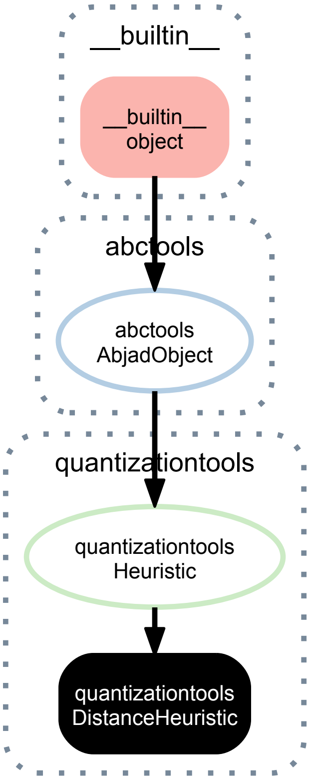
`DiscardingGraceHandler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`DiscardingGraceHandler.__repr__()`
Interpreter representation of Abjad object.

Return string.

22.2.7 quantizationtools.DistanceHeuristic



class `quantizationtools.DistanceHeuristic`
Concrete `Heuristic` subclass which considers only the computed distance of each `QGrid` and the number of leaves of that `QGrid` when choosing the optimal `QGrid` for a given `QTargetBeat`.

The `QGrid` with the smallest distance and fewest number of leaves will be selected.

Return `DistanceHeuristic` instance.

Read-only properties

`DistanceHeuristic.storage_format`
Storage format of Abjad object.

Return string.

Special methods

DistanceHeuristic.**__call__**(*q_target_beats*)

DistanceHeuristic.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DistanceHeuristic.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DistanceHeuristic.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DistanceHeuristic.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DistanceHeuristic.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

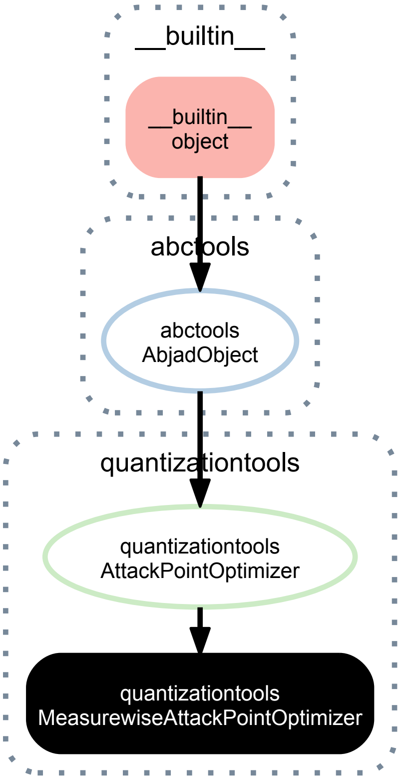
DistanceHeuristic.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DistanceHeuristic.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.8 quantizationtools.MeasurewiseAttackPointOptimizer



class `quantizationtools.`**`MeasurewiseAttackPointOptimizer`**

Concrete `AttackPointOptimizer` instance which attempts to optimize attack points in an expression with regard to the effective time signature of that expression.

Only acts on Measure instances.

Return `MeasurewiseAttackPointOptimizer` instance.

Read-only properties

`MeasurewiseAttackPointOptimizer.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`MeasurewiseAttackPointOptimizer.`**`__call__`**(*expr*)

`MeasurewiseAttackPointOptimizer.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`MeasurewiseAttackPointOptimizer.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`MeasurewiseAttackPointOptimizer.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

MeasurewiseAttackPointOptimizer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MeasurewiseAttackPointOptimizer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

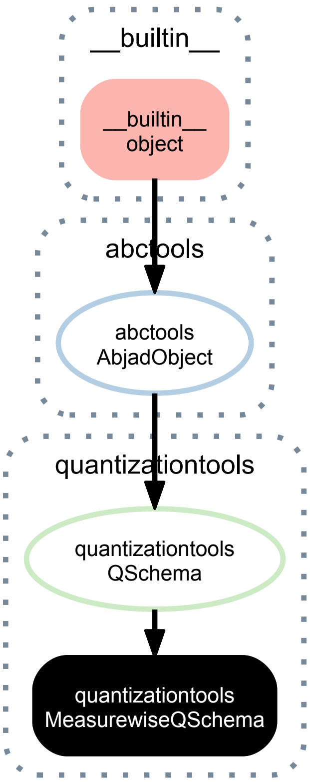
MeasurewiseAttackPointOptimizer.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MeasurewiseAttackPointOptimizer.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.9 quantizationtools.MeasurewiseQSchema



class quantizationtools.**MeasurewiseQSchema**(**args*, ***kwargs*)
Concrete QSchema subclass which treats "measures" as its time-step unit:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema()
```

Without arguments, it uses smart defaults:

```
>>> q_schema
quantizationtools.MeasurewiseQSchema(
    search_tree=quantizationtools.UnweightedSearchTree(
        definition={   2: {   2: {   2: {   2: None}, 3: None}, 3: None, 5: None, 7: None},
            3: {   2: {   2: None}, 3: None, 5: None},
            5: {   2: None, 3: None},
            7: {   2: None},
            11: None,
            13: None}
```

```
        ),
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
        ),
    time_signature=contexttools.TimeSignatureMark(
        (4, 4)
        ),
    use_full_measure=False,
    )
```

Each time-step in a `MeasurewiseQSchema` is composed of four settings:

- `search_tree`
- `tempo`
- `time_signature`
- `use_full_measure`

These settings can be applied as global defaults for the schema via keyword arguments, which persist until overridden:

```
>>> search_tree = quantizationtools.UnweightedSearchTree({7: None})
>>> time_signature = contexttools.TimeSignatureMark((3, 4))
>>> tempo = contexttools.TempoMark((1, 4), 54)
>>> use_full_measure = True
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
...     use_full_measure=use_full_measure,
...     )
```

All of these settings are self-descriptive, except for `use_full_measure`, which controls whether the measure is subdivided by the `Quantizer` into beats according to its time signature.

If `use_full_measure` is `False`, the time-step's measure will be divided into units according to its time-signature. For example, a 4/4 measure will be divided into 4 units, each having a beatspan of 1/4.

On the other hand, if `use_full_measure` is set to `True`, the time-step's measure will not be subdivided into independent quantization units. This usually results in full-measure tuplets.

The computed value at any non-negative time-step can be found by subscripting:

```
>>> index = 0
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
search_tree: UnweightedSearchTree(
    definition={   7: None}
    )
tempo: TempoMark(Duration(1, 4), 54)
time_signature: 3/4
use_full_measure: True
```

```
>>> index = 1000
>>> for key, value in sorted(q_schema[index].items()): print '{}:'.format(key), value
...
search_tree: UnweightedSearchTree(
    definition={   7: None}
    )
tempo: TempoMark(Duration(1, 4), 54)
time_signature: 3/4
use_full_measure: True
```

Per-time-step settings can be applied in a variety of ways.

Instantiating the schema via `*args` with a series of either `MeasurewiseQSchemaItem` instances, or dictionaries which could be used to instantiate `MeasurewiseQSchemaItem` instances, will apply those settings sequentially, starting from time-step `0`:

```
>>> a = {'search_tree': quantizationtools.UnweightedSearchTree({2: None})}
>>> b = {'search_tree': quantizationtools.UnweightedSearchTree({3: None})}
>>> c = {'search_tree': quantizationtools.UnweightedSearchTree({5: None})}
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(a, b, c)
```

```
>>> q_schema[0]['search_tree']
UnweightedSearchTree(
    definition={   2: None}
    )
```

```
>>> q_schema[1]['search_tree']
UnweightedSearchTree(
    definition={   3: None}
    )
```

```
>>> q_schema[2]['search_tree']
UnweightedSearchTree(
    definition={   5: None}
    )
```

```
>>> q_schema[1000]['search_tree']
UnweightedSearchTree(
    definition={   5: None}
    )
```

Similarly, instantiating the schema from a single dictionary, consisting of integer:specification pairs, or a sequence via `*args` of (integer, specification) pairs, allows for applying settings to non-sequential time-steps:

```
>>> a = {'time_signature': contexttools.TimeSignatureMark((7, 32))}
>>> b = {'time_signature': contexttools.TimeSignatureMark((3, 4))}
>>> c = {'time_signature': contexttools.TimeSignatureMark((5, 8))}
```

```
>>> settings = {
...     2: a,
...     4: b,
...     6: c,
... }
```

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(settings)
```

```
>>> q_schema[0]['time_signature']
TimeSignatureMark((4, 4))
```

```
>>> q_schema[1]['time_signature']
TimeSignatureMark((4, 4))
```

```
>>> q_schema[2]['time_signature']
TimeSignatureMark((7, 32))
```

```
>>> q_schema[3]['time_signature']
TimeSignatureMark((7, 32))
```

```
>>> q_schema[4]['time_signature']
TimeSignatureMark((3, 4))
```

```
>>> q_schema[5]['time_signature']
TimeSignatureMark((3, 4))
```

```
>>> q_schema[6]['time_signature']
TimeSignatureMark((5, 8))
```

```
>>> q_schema[1000]['time_signature']
TimeSignatureMark((5, 8))
```

The following is equivalent to the above schema definition:

```
>>> q_schema = quantizationtools.MeasurewiseQSchema(
...     (2, {'time_signature': contexttools.TimeSignatureMark((7, 32))}),
...     (4, {'time_signature': contexttools.TimeSignatureMark((3, 4))}),
...     (6, {'time_signature': contexttools.TimeSignatureMark((5, 8))}),
...     )
```

Return `MeasurewiseQSchema` instance.

Read-only properties

`MeasurewiseQSchema.`**`item_klass`**
The schema's item class.

`MeasurewiseQSchema.`**`items`**
The item dictionary.

`MeasurewiseQSchema.`**`search_tree`**
The default search tree.

`MeasurewiseQSchema.`**`storage_format`**
Storage format of Abjad object.

Return string.

`MeasurewiseQSchema.`**`target_item_klass`**

`MeasurewiseQSchema.`**`target_klass`**

`MeasurewiseQSchema.`**`tempo`**
The default tempo.

`MeasurewiseQSchema.`**`time_signature`**
The default time signature.

`MeasurewiseQSchema.`**`use_full_measure`**
The full-measure-as-beatspan default.

Special methods

`MeasurewiseQSchema.`**`__call__`**(*duration*)

`MeasurewiseQSchema.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MeasurewiseQSchema.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MeasurewiseQSchema.`**`__getitem__`**(*i*)

`MeasurewiseQSchema.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`MeasurewiseQSchema.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MeasurewiseQSchema.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

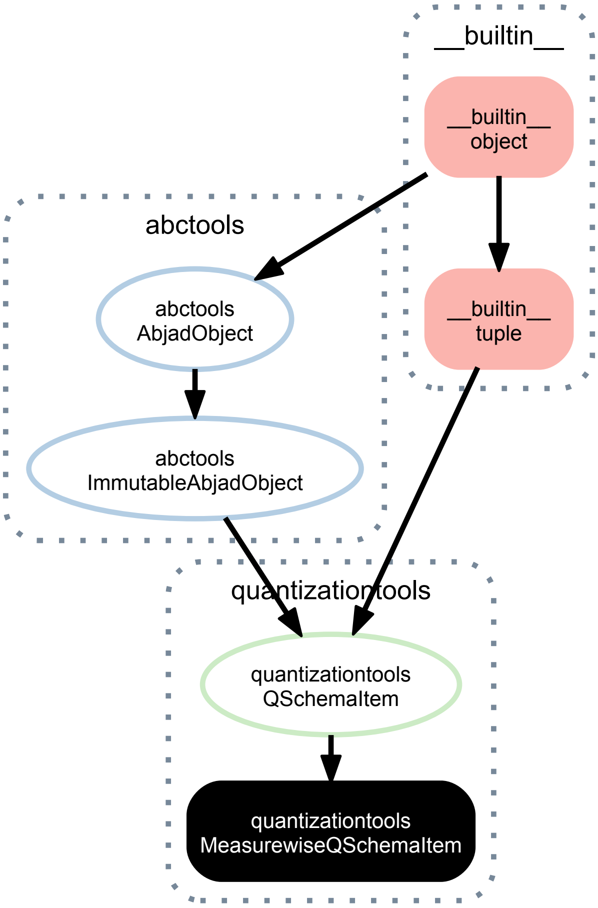
Raise exception.

MeasurewiseQSchema.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MeasurewiseQSchema.**__repr__**()

22.2.10 quantizationtools.MeasurewiseQSchemaItem



class quantizationtools.**MeasurewiseQSchemaItem**(**args*, ***kwargs*)
MeasurewiseQSchemaItem represents a change of state in the timeline of a metered quantization process.

```
>>> quantizationtools.MeasurewiseQSchemaItem()
MeasurewiseQSchemaItem()
```

Define a change in tempo:

```
>>> quantizationtools.MeasurewiseQSchemaItem(tempo=((1, 4), 60))
MeasurewiseQSchemaItem(
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
        ),
    )
```

Define a change in time signature:

```
>>> quantizationtools.MeasurewiseQSchemaItem(time_signature=((6, 8)))
MeasurewiseQSchemaItem(
    time_signature=contexttools.TimeSignatureMark(
        (6, 8)
        ),
    )
```

Test for beatspan, given a defined time signature:

```
>>> _.beatspan
Duration(1, 8)
```

MeasurewiseQSchemaItem is immutable.

Return *MeasurewiseQSchemaItem* instance.

Read-only properties

`MeasurewiseQSchemaItem.`**`beatspan`**
The beatspan duration, if a time signature was defined.

Return *Duration* or *None*.

`MeasurewiseQSchemaItem.`**`search_tree`**
The optionally defined search tree.

Return *OldSearchTree* or *None*.

`MeasurewiseQSchemaItem.`**`storage_format`**
Storage format of Abjad object.

Return string.

`MeasurewiseQSchemaItem.`**`tempo`**
The optionally defined *TempoMark*.

Return *TempoMark* or *None*.

`MeasurewiseQSchemaItem.`**`time_signature`**
The optionally defined TimeSignatureMark.

Return *TimeSignatureMark* or None.

`MeasurewiseQSchemaItem.`**`use_full_measure`**
If True, use the full measure as the beatspan.

Return bool or None.

Methods

`MeasurewiseQSchemaItem.`**`count`**(*value*) → integer – return number of occurrences of value

`MeasurewiseQSchemaItem.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`MeasurewiseQSchemaItem.`**`__add__`**()
x.__add__(y) <==> x+y

`MeasurewiseQSchemaItem.`**`__contains__`**()
x.__contains__(y) <==> y in x

`MeasurewiseQSchemaItem.`**`__eq__`**()
x.__eq__(y) <==> x==y

`MeasurewiseQSchemaItem.`**`__ge__`**()
x.__ge__(y) <==> x>=y

`MeasurewiseQSchemaItem.`**`__getitem__`**()
x.__getitem__(y) <==> x[y]

MeasurewiseQSchemaItem.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

MeasurewiseQSchemaItem.**__gt__**()
x.__gt__(y) <==> x>y

MeasurewiseQSchemaItem.**__hash__**() <==> *hash(x)*

MeasurewiseQSchemaItem.**__iter__**() <==> *iter(x)*

MeasurewiseQSchemaItem.**__le__**()
x.__le__(y) <==> x<=y

MeasurewiseQSchemaItem.**__len__**() <==> *len(x)*

MeasurewiseQSchemaItem.**__lt__**()
x.__lt__(y) <==> x<y

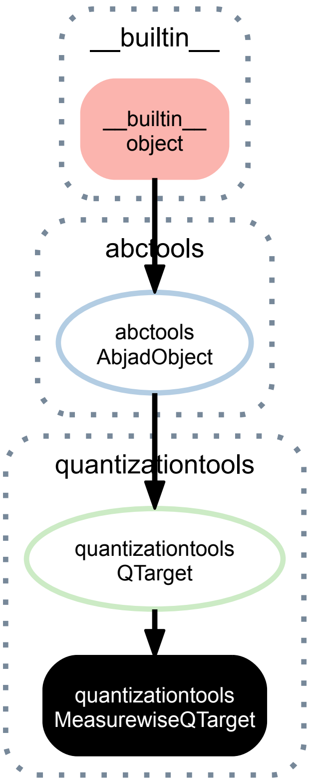
MeasurewiseQSchemaItem.**__mul__**()
x.__mul__(n) <==> x*n

MeasurewiseQSchemaItem.**__ne__**()
x.__ne__(y) <==> x!=y

MeasurewiseQSchemaItem.**__repr__**()

MeasurewiseQSchemaItem.**__rmul__**()
x.__rmul__(n) <==> n*x

22.2.11 quantizationtools.MeasurewiseQTarget



class quantizationtools.**MeasurewiseQTarget**(*items*)
A measure-wise quantization target.

Not composer-safe.

Used internally by `Quantizer`.

Read-only properties

`MeasurewiseQTarget.`**`beats`**

`MeasurewiseQTarget.`**`duration_in_ms`**

`MeasurewiseQTarget.`**`item_klass`**

`MeasurewiseQTarget.`**`items`**

`MeasurewiseQTarget.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`MeasurewiseQTarget.`**`__call__`**(*q_event_sequence, grace_handler=None, heuristic=None, job_handler=None, attack_point_optimizer=None, attach_tempo_marks=True*)

`MeasurewiseQTarget.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MeasurewiseQTarget.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MeasurewiseQTarget.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`MeasurewiseQTarget.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MeasurewiseQTarget.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

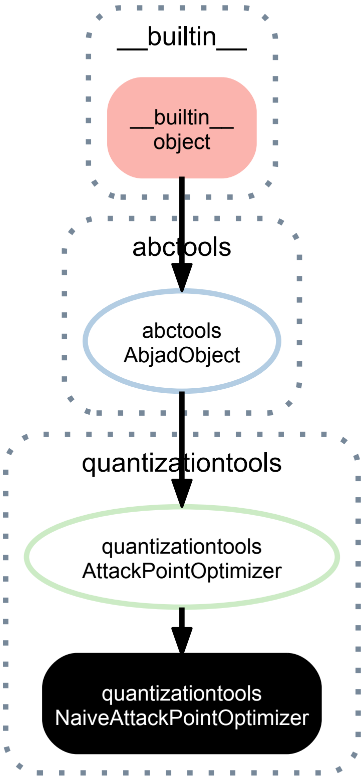
`MeasurewiseQTarget.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`MeasurewiseQTarget.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

22.2.12 quantizationtools.NaiveAttackPointOptimizer



class `quantizationtools.`**`NaiveAttackPointOptimizer`**

Concrete `AttackPointOptimizer` subclass which optimizes attack points by fusing tie leaves within tie chains with leaf durations decreasing monotonically.

`TieChains` will be partitioned into sub-`TieChains` if leaves are found with `TempoMarks` attached.

Return `NaiveAttackPointOptimizer` instance.

Read-only properties

`NaiveAttackPointOptimizer.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`NaiveAttackPointOptimizer.`**`__call__`**(*expr*)

`NaiveAttackPointOptimizer.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`NaiveAttackPointOptimizer.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`NaiveAttackPointOptimizer.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

NaiveAttackPointOptimizer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NaiveAttackPointOptimizer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

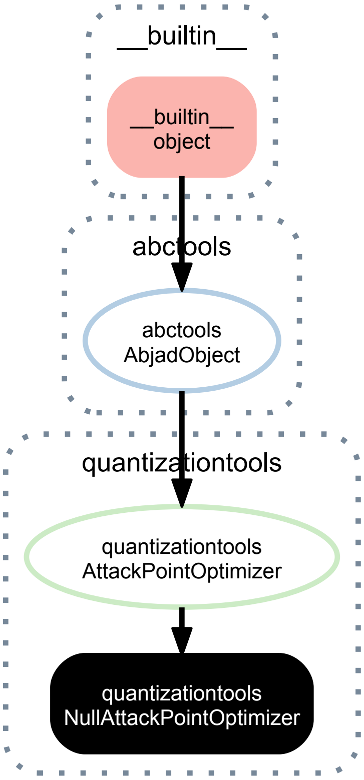
NaiveAttackPointOptimizer.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

NaiveAttackPointOptimizer.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.13 quantizationtools.NullAttackPointOptimizer



class quantizationtools.**NullAttackPointOptimizer**
Concrete `AttackPointOptimizer` subclass which performs no attack point optimization.

Return `NullAttackPointOptimizer` instance.

Read-only properties

NullAttackPointOptimizer.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

NullAttackPointOptimizer.**__call__**(*expr*)

NullAttackPointOptimizer.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

NullAttackPointOptimizer.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NullAttackPointOptimizer.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

NullAttackPointOptimizer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NullAttackPointOptimizer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

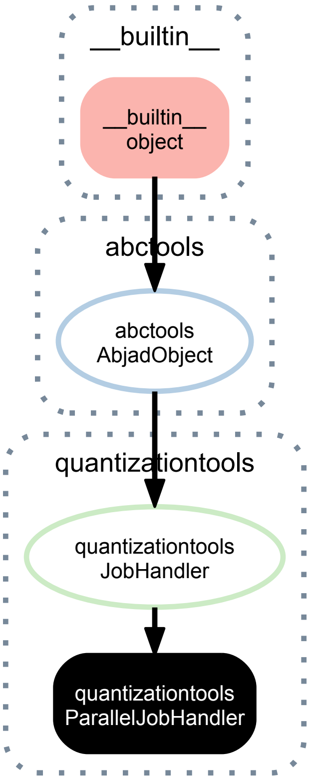
NullAttackPointOptimizer.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

NullAttackPointOptimizer.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.14 quantizationtools.ParallelJobHandler



class `quantizationtools.`**`ParallelJobHandler`**

Processes `QuantizationJob` instances in parallel, based on the number of CPUs available.

Read-only properties

`ParallelJobHandler.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`ParallelJobHandler.`**`__call__`**(*jobs*)

`ParallelJobHandler.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`ParallelJobHandler.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`ParallelJobHandler.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`ParallelJobHandler.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ParallelJobHandler.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

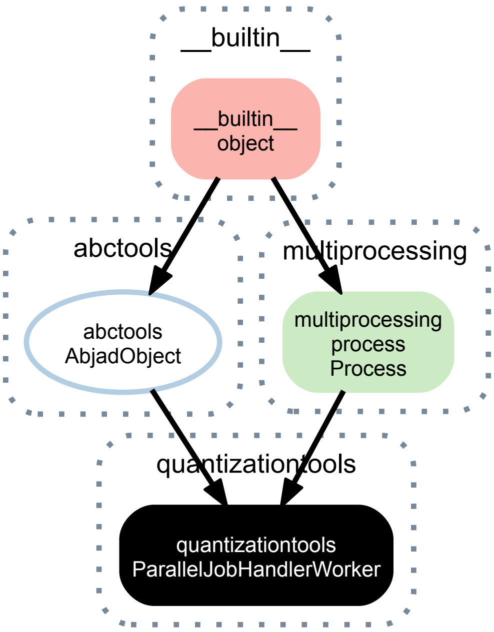
ParallelJobHandler.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ParallelJobHandler.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.15 quantizationtools.ParallelJobHandlerWorker



class quantizationtools.**ParallelJobHandlerWorker**(*job_queue*, *result_queue*)
Worker process which runs QuantizationJobs.

Not composer-safe.

Used internally by ParallelJobHandler.

Return ParallelJobHandlerWorker instance.

Read-only properties

ParallelJobHandlerWorker.**exitcode**
Return exit code of process or *None* if it has yet to stop

ParallelJobHandlerWorker.**ident**
Return identifier (PID) of process or *None* if it has yet to start

ParallelJobHandlerWorker.**pid**
Return identifier (PID) of process or *None* if it has yet to start

ParallelJobHandlerWorker.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

`ParallelJobHandlerWorker.`**authkey**

`ParallelJobHandlerWorker.`**daemon**
Return whether process is a daemon

`ParallelJobHandlerWorker.`**name**

Methods

`ParallelJobHandlerWorker.`**is_alive**()
Return whether process is alive

`ParallelJobHandlerWorker.`**join**(*timeout=None*)
Wait until child process terminates

`ParallelJobHandlerWorker.`**run**()

`ParallelJobHandlerWorker.`**start**()
Start child process

`ParallelJobHandlerWorker.`**terminate**()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

Special methods

`ParallelJobHandlerWorker.`**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`ParallelJobHandlerWorker.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ParallelJobHandlerWorker.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`ParallelJobHandlerWorker.`**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ParallelJobHandlerWorker.`**__lt__**(*expr*)
Abjad objects by default do not implement this method.

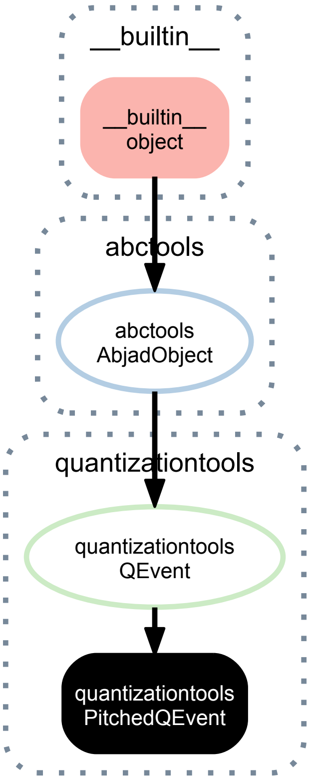
Raise exception.

`ParallelJobHandlerWorker.`**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`ParallelJobHandlerWorker.`**__repr__**()

22.2.16 quantizationtools.PitchedQEvent



class quantizationtools.**PitchedQEvent** (*offset*, *pitches*, *attachments=None*, *index=None*)

A QEvent which indicates the onset of a period of pitched material in a QEventSequence:

```
>>> pitches = [0, 1, 4]
>>> q_event = quantizationtools.PitchedQEvent(1000, pitches)
>>> q_event
quantizationtools.PitchedQEvent(
    durationtools.Offset(1000, 1),
    (NamedChromaticPitch("c'"), NamedChromaticPitch("cs'"), NamedChromaticPitch("e'")),
    attachments=()
    )
```

Return PitchedQEvent instance.

Read-only properties

PitchedQEvent.**attachments**

PitchedQEvent.**index**

The optional index, for sorting QEvents with identical offsets.

PitchedQEvent.**offset**

The offset in milliseconds of the event.

PitchedQEvent.**pitches**

PitchedQEvent.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

PitchedQEvent.**__eq__** (*expr*)

`PitchedQEvent.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PitchedQEvent.__getstate__()`

`PitchedQEvent.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`PitchedQEvent.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PitchedQEvent.__lt__(expr)`

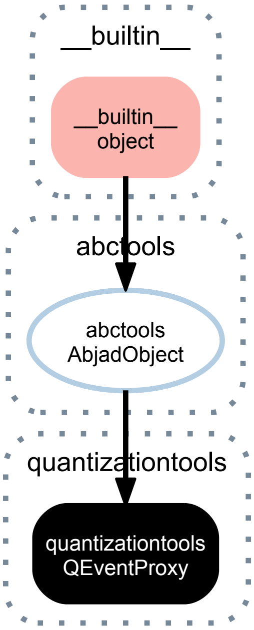
`PitchedQEvent.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`PitchedQEvent.__repr__()`

`PitchedQEvent.__setstate__(state)`

22.2.17 quantizationtools.QEventProxy



class `quantizationtools.QEventProxy(*args)`
Proxies a *QEvent*, mapping that QEvent's offset with the range of its beatspan to the range 0-1:

```
>>> q_event = quantizationtools.PitchedQEvent(130, [0, 1, 4])
>>> proxy = quantizationtools.QEventProxy(q_event, 0.5)
>>> proxy
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(130, 1),
        (NamedChromaticPitch("c'"), NamedChromaticPitch("cs'"), NamedChromaticPitch("e'")),
        attachments=()
        ),
    durationtools.Offset(1, 2)
    )
```

Not composer-safe.

Used internally by `Quantizer`.

Returns *QEventProxy* instance.

Read-only properties

`QEventProxy.`**`index`**

`QEventProxy.`**`offset`**

`QEventProxy.`**`q_event`**

`QEventProxy.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`QEventProxy.`**`__eq__`**`(`*expr*`)`

`QEventProxy.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`QEventProxy.`**`__getstate__`**`()`

`QEventProxy.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`QEventProxy.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`QEventProxy.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

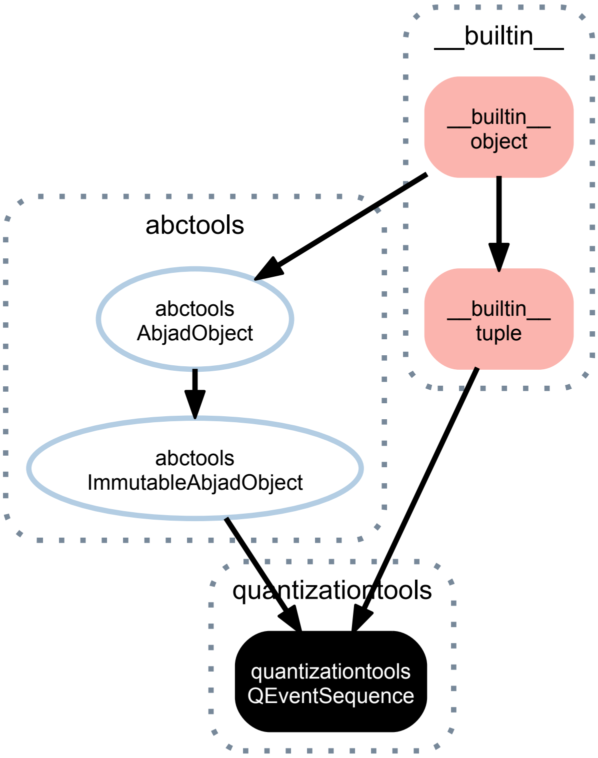
`QEventProxy.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`QEventProxy.`**`__repr__`**`()`

`QEventProxy.`**`__setstate__`**`(`*state*`)`

22.2.18 quantizationtools.QEventSequence



class `quantizationtools.QEventSequence`(**args*, ***kwargs*)

A well-formed sequence of `QEvent` instances, containing only `PitchedQEvents` and `SilentQEvents`, and terminating with a single `TerminalQEvent` instance.

`QEventSequence` is the primary input to the `Quantizer`.

`QEventSequence` provides a number of convenience functions to assist with instantiating new sequences:

```
>>> durations = (1000, -500, 1250, -500, 750)
```

```
>>> sequence = quantizationtools.QEventSequence.from_millisecond_durations(
...     durations)
```

```
>>> for q_event in sequence:
...     q_event
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(1000, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1500, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(2750, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(3250, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.TerminalQEvent(
```

```
        durationtools.Offset(4000, 1)
        )
```

Return `QEventSequence` instance.

Read-only properties

`QEventSequence.`**`duration_in_ms`**
The total duration in milliseconds of the `QEventSequence`:

```
>>> sequence.duration_in_ms
Duration(4000, 1)
```

Return `Duration` instance.

`QEventSequence.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`QEventSequence.`**`count`**(*value*) → integer – return number of occurrences of value

classmethod `QEventSequence.`**`from_millisecond_durations`**(*klass*, *durations*, *fuse_silences=False*)
Convert a sequence of millisecond durations `durations` into a `QEventSequence`:

```
>>> durations = [-250, 500, -1000, 1250, -1000]
```

```
>>> sequence = quantizationtools.QEventSequence.from_millisecond_durations(
...     durations)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.SilentQEvent(
    durationtools.Offset(0, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(250, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(750, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1750, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(3000, 1),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
    )
```

Return `QEventSequence` instance.

classmethod `QEventSequence.`**`from_millisecond_offsets`**(*klass*, *offsets*)
Convert millisecond offsets `offsets` into a `QEventSequence`:

```
>>> offsets = [0, 250, 750, 1750, 3000, 4000]
```

```
>>> sequence = quantizationtools.QEventSequence.from_millisecond_offsets(
...     offsets)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(250, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(750, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1750, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(3000, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
    )
```

Return `QEventSequence` instance.

classmethod `QEventSequence.`**`from_millisecond_pitch_pairs`**(*klass*, *pairs*)
Convert millisecond-duration:pitch pairs `pairs` into a `QEventSequence`:

```
>>> durations = [250, 500, 1000, 1250, 1000]
>>> pitches = [(0,), None, (2, 3), None, (1,)]
>>> pairs = zip(durations, pitches)
```

```
>>> sequence = quantizationtools.QEventSequence.from_millisecond_pitch_pairs(
...     pairs)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(250, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(750, 1),
    (NamedChromaticPitch("d'"), NamedChromaticPitch("ef'")),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(1750, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
```

```
        durationtools.Offset(3000, 1),
        (NamedChromaticPitch("cs''"),),
        attachments=()
        )
    quantizationtools.TerminalQEvent(
        durationtools.Offset(4000, 1)
        )
```

Return `QEventSequence` instance.

classmethod `QEventSequence.`**`from_tempo_scaled_durations`**(*klass*, *durations*, *tempo=None*)

Convert `durations`, scaled by `tempo` into a `QEventSequence`:

```
>>> tempo = contexttools.TempoMark((1, 4), 174)
>>> durations = [(1, 4), (-3, 16), (1, 16), (-1, 2)]
```

```
>>> sequence = quantizationtools.QEventSequence.from_tempo_scaled_durations(
...     durations, tempo=tempo)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(10000, 29),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(17500, 29),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(20000, 29),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(40000, 29)
    )
```

Return `QEventSequence` instance.

classmethod `QEventSequence.`**`from_tempo_scaled_leaves`**(*klass*, *leaves*, *tempo=None*)

Convert `leaves`, optionally with `tempo` into a `QEventSequence`:

```
>>> staff = Staff("c'4 <d' fs'>8. r16 gqs'2")
>>> tempo = contexttools.TempoMark((1, 4), 72)
```

```
>>> sequence = quantizationtools.QEventSequence.from_tempo_scaled_leaves(
...     staff.leaves, tempo=tempo)
```

```
>>> for q_event in sequence:
...     q_event
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(2500, 3),
    (NamedChromaticPitch("d'"), NamedChromaticPitch("fs'")),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(4375, 3),
```

```
        attachments=()
        )
    quantizationtools.PitchedQEvent(
        durationtools.Offset(5000, 3),
        (NamedChromaticPitch("gqs'"),),
        attachments=()
        )
    quantizationtools.TerminalQEvent(
        durationtools.Offset(10000, 3)
        )
```

If `tempo` is `None`, all leaves in `leaves` must have an effective, non-imprecise tempo. The millisecond-duration of each leaf will be determined by its effective tempo.

Return `QEventSequence` instance.

`QEventSequence.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`QEventSequence.`**`__add__`**()
x.__add__(y) <==> x+y

`QEventSequence.`**`__contains__`**()
x.__contains__(y) <==> y in x

`QEventSequence.`**`__eq__`**()
x.__eq__(y) <==> x==y

`QEventSequence.`**`__ge__`**()
x.__ge__(y) <==> x>=y

`QEventSequence.`**`__getitem__`**()
x.__getitem__(y) <==> x[y]

`QEventSequence.`**`__getslice__`**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`QEventSequence.`**`__gt__`**()
x.__gt__(y) <==> x>y

`QEventSequence.`**`__hash__`**() <==> *hash(x)*

`QEventSequence.`**`__iter__`**() <==> *iter(x)*

`QEventSequence.`**`__le__`**()
x.__le__(y) <==> x<=y

`QEventSequence.`**`__len__`**() <==> *len(x)*

`QEventSequence.`**`__lt__`**()
x.__lt__(y) <==> x<y

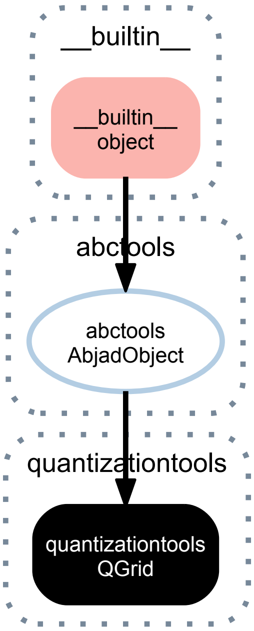
`QEventSequence.`**`__mul__`**()
x.__mul__(n) <==> x*n

`QEventSequence.`**`__ne__`**()
x.__ne__(y) <==> x!=y

`QEventSequence.`**`__repr__`**()

`QEventSequence.`**`__rmul__`**()
x.__rmul__(n) <==> n*x

22.2.19 quantizationtools.QGrid



class `quantizationtools.QGrid`(*root_node=None*, *next_downbeat=None*)

A rhythm-tree-based model for how millisecond attack points collapse onto the offsets generated by a nested rhythmic structure:

```
>>> q_grid = quantizationtools.QGrid()
```

```
>>> q_grid
quantizationtools.QGrid(
    root_node=quantizationtools.QGridLeaf(
        preprolated_duration=durationtools.Duration(1, 1),
        q_event_proxies=[],
        is_divisible=True
        ),
    next_downbeat=quantizationtools.QGridLeaf(
        preprolated_duration=durationtools.Duration(1, 1),
        q_event_proxies=[],
        is_divisible=True
        )
    )
```

`QGrids` model not only the internal nodes of the nesting structure, but also the downbeat to the "next" `QGrid`, allowing events which occur very late within one structure to collapse virtually onto the beginning of the next structure.

`QEventProxies` can be "loaded in" to the node contained by the `QGrid` closest to their virtual offset:

```
>>> q_event_a = quantizationtools.PitchedQEvent(250, [0])
>>> q_event_b = quantizationtools.PitchedQEvent(750, [1])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.25)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.75)
```

```
>>> q_grid.fit_q_events([proxy_a, proxy_b])
```

```
>>> q_grid.root_node.q_event_proxies
[quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(250, 1),
        (NamedChromaticPitch("c'"),),
        attachments=()
        ),
    durationtools.Offset(1, 4)
    )]
```

```
>>> q_grid.next_downbeat.q_event_proxies
[quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(750, 1),
        (NamedChromaticPitch("cs'"),),
        attachments=()
        ),
    durationtools.Offset(3, 4)
    )]
```

Used internally by the `Quantizer`.

Return `QGrid` instance.

Read-only properties

`QGrid.`**`distance`**
The computed total distance of the offset of each `QEventProxy` contained by the `QGrid` to the offset of the `QGridLeaf` to which the `QEventProxy` is attached.

Return `Duration` instance.

`QGrid.`**`leaves`**
All of the leaf nodes in the QGrid, including the next downbeat's node.

Return tuple of `QGridLeaf` instances.

`QGrid.`**`next_downbeat`**
The node representing the "next" downbeat after the contents of the QGrid's tree.

Return `QGridLeaf` instance.

`QGrid.`**`offsets`**
The offsets between 0 and 1 of all of the leaf nodes in the QGrid.

Return tuple of `Offset` instances.

`QGrid.`**`pretty_rtm_format`**
The pretty RTM-format of the root node of the `QGrid`.

Return string.

`QGrid.`**`root_node`**
The root node of the `QGrid`.

Return `QGridLeaf` or `QGridContainer`.

`QGrid.`**`rtm_format`**
The RTM format of the root node of the `QGrid`.

Return string.

`QGrid.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`QGrid.`**`fit_q_events`**`(`*q_event_proxies*`)`
Fit each `QEventProxy` in `q_event_proxies` onto the contained `QGridLeaf` whose offset is nearest.

Return None

`QGrid.`**`sort_q_events_by_index`**`()`
Sort `QEventProxies` attached to each `QGridLeaf` in a `QGrid` by their index.

Return None.

QGrid.**subdivide_leaf**(*leaf*, *subdivisions*)
Replace the `QGridLeaf leaf` contained in a `QGrid` by a `QGridContainer` containing `QGridLeaves` with durations equal to the ratio described in `subdivisions`

Return the `QEventProxies` attached to `leaf`.

QGrid.**subdivide_leaves**(*pairs*)
Given a sequence of leaf-index:subdivision-ratio pairs `pairs`, subdivide the `QGridLeaves` described by the indices into `QGridContainers` containing `QGridLeaves` with durations equal to their respective subdivision-ratios.

Return the `QEventProxies` attached to thus subdivided `QGridLeaf`.

Special methods

QGrid.**__call__**(*beatspan*)

QGrid.**__copy__**(**args*)

QGrid.**__deepcopy__**(*memo*)

QGrid.**__eq__**(*expr*)

QGrid.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QGrid.**__getstate__**()

QGrid.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

QGrid.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QGrid.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

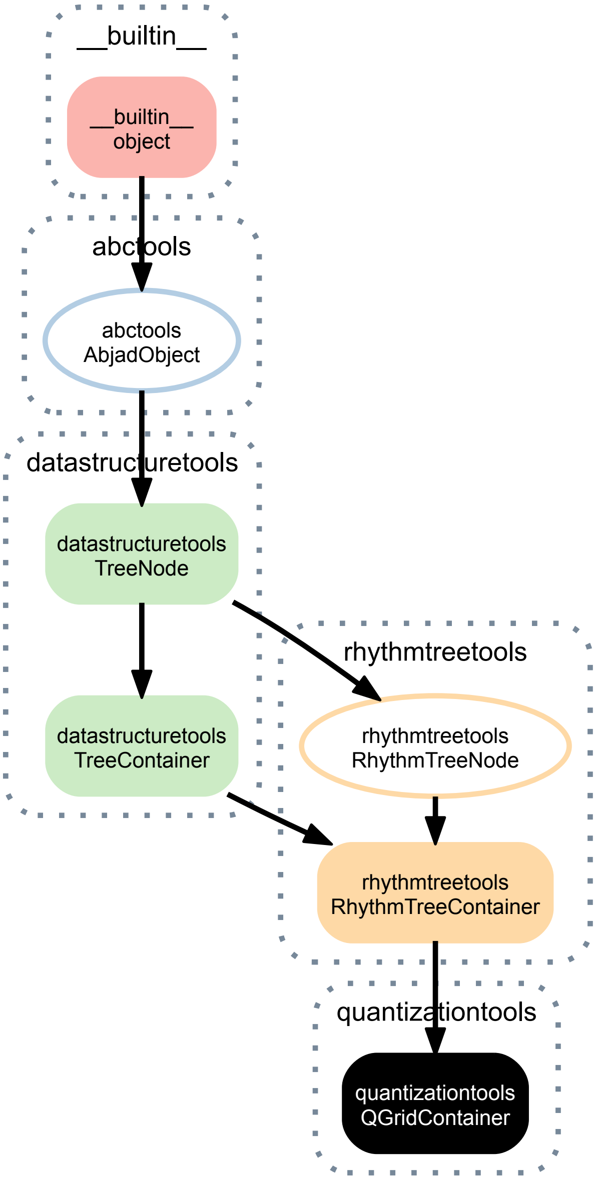
QGrid.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

QGrid.**__repr__**()

QGrid.**__setstate__**(*state*)

22.2.20 quantizationtools.QGridContainer



class `quantizationtools.`**`QGridContainer`**(*children=None*, *preprolated_duration=1*, *name=None*)

A container in a `QGrid` structure:

```
>>> container = quantizationtools.QGridContainer()
```

```
>>> container
QGridContainer(
    preprolated_duration=Duration(1, 1)
    )
```

Used internally by `QGrid`.

Return `QGridContainer` instance.

Read-only properties

`QGridContainer.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

QGridContainer.**contents_duration**

The total preprolated_duration of the children of a *RhythmTreeContainer* instance:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.contents_duration
Duration(5, 1)
```

```
>>> tree[1].contents_duration
Duration(3, 1)
```

Return int.

QGridContainer.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

QGridContainer.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

QGridContainer.**duration**
The prolated preprolated_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

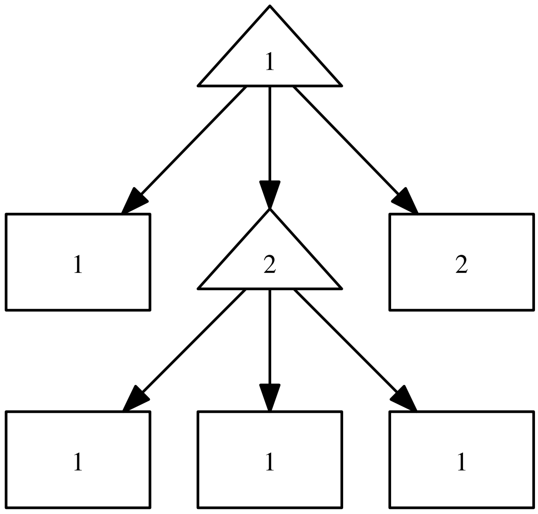
QGridContainer.**graph_order**

QGridContainer.**graphviz_format**

QGridContainer.**graphviz_graph**
The GraphvizGraph representation of the RhythmTreeContainer:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> graph = tree.graphviz_graph
>>> print graph.graphviz_format
digraph G {
    node_0 [label=1,
        shape=triangle];
    node_1 [label=1,
        shape=box];
    node_2 [label=2,
        shape=triangle];
    node_3 [label=1,
        shape=box];
    node_4 [label=1,
        shape=box];
    node_5 [label=1,
        shape=box];
    node_6 [label=2,
        shape=box];
    node_0 -> node_1;
    node_0 -> node_2;
    node_0 -> node_6;
    node_2 -> node_3;
    node_2 -> node_4;
    node_2 -> node_5;
}
```

```
>>> iotools.graph(graph)
```



Return *GraphvizGraph* instance.

QGridContainer.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

QGridContainer.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

QGridContainer.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`QGridContainer.`**`parent`**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`QGridContainer.`**`parentage_ratios`**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated_duration of the root node, and subsequent items are pairs of the preprolated_duration of the next node in the parentage chain and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Return tuple.

`QGridContainer.`**`pretty_rtm_format`**
The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
    (1 (
        1
        1))
    (1 (
        1
        1))))
```

Return string.

`QGridContainer.`**`prolation`**

`QGridContainer.`**`prolations`**

`QGridContainer.`**`proper_parentage`**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

`QGridContainer.`**`root`**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

QGridContainer.**rtm_format**
The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Return string.

QGridContainer.**start_offset**
The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Return Offset instance.

QGridContainer.**stop_offset**
The stopping offset of a node in a rhythm-tree relative the root.

QGridContainer.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

QGridContainer.**name**

QGridContainer.**preprolated_duration**
The node's preprolated_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Return int.

Methods

QGridContainer.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`QGridContainer.`**`extend`**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`QGridContainer.`**`index`**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`QGridContainer.`**`insert`**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
```

```
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`QGridContainer.`**`pop`**`(`*i=-1*`)`

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

`QGridContainer.`**`remove`**`(`*node*`)`

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
```

```
            TreeNode()
            )
        )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

`QGridContainer.__add__(expr)`

Concatenate containers self and expr. The operation c = a + b returns a new RhythmTreeContainer c with the content of both a and b, and a preprolated_duration equal to the sum of the durations of a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand:

```
>>> a = rhythmtreetools.RhythmTreeParser()('(1 (1 1 1))')[0]
>>> b = rhythmtreetools.RhythmTreeParser()('(2 (3 4))')[0]
```

```
>>> c = a + b
```

```
>>> c.preprolated_duration
Duration(3, 1)
```

```
>>> c
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(3, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(4, 1),
            is_pitched=True
            )
        ),
    preprolated_duration=Duration(3, 1)
    )
```

Return new RhythmTreeContainer.

`QGridContainer.__call__(pulse_duration)`

Generate Abjad score components:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree((1, 4))
[FixedDurationTuplet(1/4, [c'16, {@ 3:2 c'16, c'16, c'16 @}, c'8])]
```

Return sequence of components.

QGridContainer.**__contains__**(*expr*)
True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

QGridContainer.**__copy__**(**args*)

QGridContainer.**__deepcopy__**(**args*)

QGridContainer.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

QGridContainer.**__eq__**(*expr*)
True if type, preprolated_duration and children are equivalent, otherwise False.

Return boolean.

QGridContainer.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QGridContainer.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

`QGridContainer.`**`__getstate__`**`()`

`QGridContainer.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`QGridContainer.`**`__iter__`**`()`

`QGridContainer.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`QGridContainer.`**`__len__`**`()`
Return nonnegative integer number of nodes in container.

`QGridContainer.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`QGridContainer.`**`__ne__`**`(`*expr*`)`

`QGridContainer.`**`__repr__`**`()`

`QGridContainer.`**`__setitem__`**`(`*i*, *expr*`)`
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = rhythmtreetools.RhythmTreeContainer()
>>> b = rhythmtreetools.RhythmTreeLeaf()
>>> c = rhythmtreetools.RhythmTreeLeaf()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

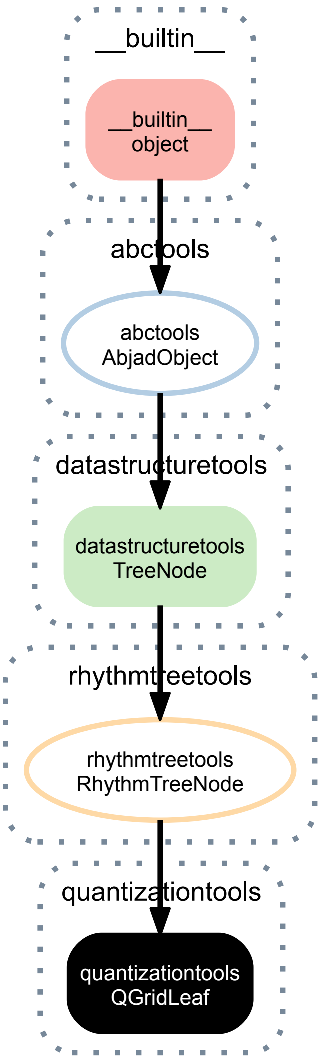
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

`QGridContainer.`**`__setstate__`**(*state*)

22.2.21 quantizationtools.QGridLeaf



class `quantizationtools.`**`QGridLeaf`**(*preprolated_duration=1*, *q_event_proxies=None*, *is_divisible=True*)

A leaf in a `QGrid` structure:

```
>>> leaf = quantizationtools.QGridLeaf()
```

```
>>> leaf
QGridLeaf(
    preprolated_duration=Duration(1, 1),
    is_divisible=True
    )
```

Used internally by `QGrid`.

Return `QGridLeaf` instance.

Read-only properties

`QGridLeaf.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`QGridLeaf.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

QGridLeaf.**duration**

The prolated preprolated_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

QGridLeaf.**graph_order**

QGridLeaf.**graphviz_format**

QGridLeaf.**graphviz_graph**

QGridLeaf.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

QGridLeaf.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

QGridLeaf.**parentage_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated_duration of the root node, and subsequent items are pairs of the preprolated_duration of the next node in the parentage chain and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Return tuple.

QGridLeaf.**preceding_q_event_proxies**

QGridLeaf.**pretty_rtm_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
    (1 (
        1
        1))
    (1 (
        1
        1))))
```

Return string.

QGridLeaf.**prolation**

QGridLeaf.**prolations**

QGridLeaf.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

QGridLeaf.**q_event_proxies**

QGridLeaf.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

QGridLeaf.**rtm_format**

QGridLeaf.**start_offset**

The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Return Offset instance.

QGridLeaf.**stop_offset**

The stopping offset of a node in a rhythm-tree relative the root.

QGridLeaf.**storage_format**

Storage format of Abjad object.

Return string.

QGridLeaf.**succeeding_q_event_proxies**

Read/write properties

QGridLeaf.**is_divisible**

Flag for whether the node may be further divided under some search tree.

QGridLeaf.**name**

QGridLeaf.**preprolated_duration**

The node's preprolated_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Return int.

Special methods

QGridLeaf.**__call__**(*pulse_duration*)

QGridLeaf.**__copy__**(**args*)

QGridLeaf.**__deepcopy__**(*memo*)

QGridLeaf.**__eq__**(*expr*)

QGridLeaf.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QGridLeaf.**__getstate__**()

QGridLeaf.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

QGridLeaf.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QGridLeaf.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

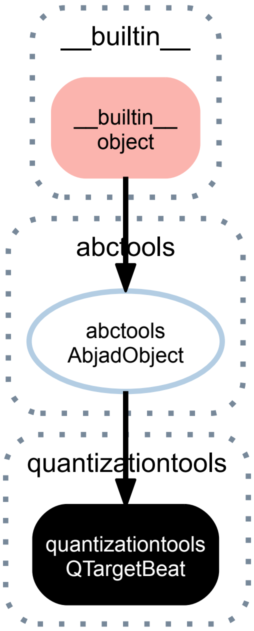
Raise exception.

QGridLeaf.**__ne__**(*expr*)

QGridLeaf.**__repr__**()

QGridLeaf.**__setstate__**(*state*)

22.2.22 quantizationtools.QTargetBeat



class `quantizationtools.`**`QTargetBeat`**(*beatspan=None*, *offset_in_ms=None*, *search_tree=None*, *tempo=None*)

Representation of a single "beat" in a quantization target:

```
>>> beatspan = (1, 8)
>>> offset_in_ms = 1500
>>> search_tree = quantizationtools.UnweightedSearchTree({3: None})
>>> tempo = contexttools.TempoMark((1, 4), 56)
```

```
>>> q_target_beat = quantizationtools.QTargetBeat(
...     beatspan=beatspan,
...     offset_in_ms=offset_in_ms,
...     search_tree=search_tree,
...     tempo=tempo,
...     )
```

```
>>> q_target_beat
quantizationtools.QTargetBeat(
    beatspan=durationtools.Duration(1, 8),
    offset_in_ms=durationtools.Offset(1500, 1),
    search_tree=quantizationtools.UnweightedSearchTree(
        definition={    3: None}
        ),
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        56
        )
    )
```

Not composer-safe.

Used internally by quantizationtools.Quantizer.

Return `QTargetBeat` instance.

Read-only properties

`QTargetBeat.`**`beatspan`**

The beatspan of the `QTargetBeat`:

```
>>> q_target_beat.beatspan
Duration(1, 8)
```

Return Duration.

QTargetBeat.**distances**

A list of computed distances between the QEventProxies associated with a QTargetBeat instance, and each QGrid generated for that beat.

Used internally by the Quantizer.

Return tuple.

QTargetBeat.**duration_in_ms**

The duration in milliseconds of the QTargetBeat:

```
>>> q_target_beat.duration_in_ms
Duration(3750, 7)
```

Return Duration instance.

QTargetBeat.**offset_in_ms**

The offset in milliseconds of the QTargetBeat:

```
>>> q_target_beat.offset_in_ms
Offset(1500, 1)
```

Return Offset instance.

QTargetBeat.**q_events**

A list for storing QEventProxy instances.

Used internally by the Quantizer.

Return list.

QTargetBeat.**q_grid**

The QGrid instance selected by a Heuristic.

Used internally by the Quantizer.

Return QGrid instance.

QTargetBeat.**q_grids**

A tuple of QGrids generated by a QuantizationJob.

Used internally by the Quantizer.

Return tuple.

QTargetBeat.**search_tree**

The search tree of the QTargetBeat:

```
>>> q_target_beat.search_tree
UnweightedSearchTree(
    definition={    3: None}
    )
```

Return SearchTree instance.

QTargetBeat.**storage_format**

Storage format of Abjad object.

Return string.

QTargetBeat.**tempo**

The tempo of the QTargetBeat:

```
>>> q_target_beat.tempo
TempoMark(Duration(1, 4), 56)
```

Return `TempoMark` instance.

Special methods

`QTargetBeat.__call__`(*job_id*)

`QTargetBeat.__eq__`(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`QTargetBeat.__ge__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`QTargetBeat.__gt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`QTargetBeat.__le__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`QTargetBeat.__lt__`(*expr*)
Abjad objects by default do not implement this method.

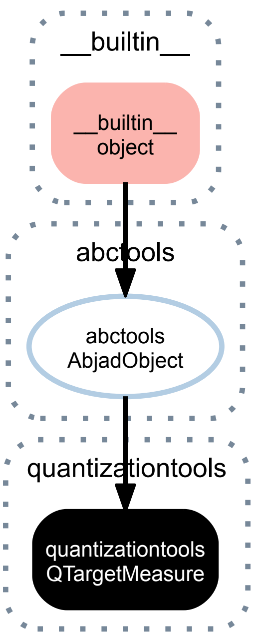
Raise exception.

`QTargetBeat.__ne__`(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`QTargetBeat.__repr__`()

22.2.23 quantizationtools.QTargetMeasure



class `quantizationtools.`**`QTargetMeasure`**(*offset_in_ms=None*, *search_tree=None*, *time_signature=None*, *tempo=None*, *use_full_measure=False*)

Representation of a single "measure" in a measure-wise quantization target.

```
>>> search_tree = quantizationtools.UnweightedSearchTree({2: None})
>>> tempo = contexttools.TempoMark((1, 4), 60)
>>> time_signature = contexttools.TimeSignatureMark((4, 4))
```

```
>>> q_target_measure = quantizationtools.QTargetMeasure(
...     offset_in_ms=1000,
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
...     )
```

```
>>> q_target_measure
quantizationtools.QTargetMeasure(
    offset_in_ms=durationtools.Offset(1000, 1),
    search_tree=quantizationtools.UnweightedSearchTree(
        definition={    2: None}
        ),
    time_signature=contexttools.TimeSignatureMark(
        (4, 4)
        ),
    tempo=contexttools.TempoMark(
        durationtools.Duration(1, 4),
        60
        ),
    use_full_measure=False
    )
```

`QTargetMeasures` group `QTargetBeats`:

```
>>> for q_target_beat in q_target_measure.beats:
...     print q_target_beat.offset_in_ms, q_target_beat.duration_in_ms
1000 1000
2000 1000
3000 1000
4000 1000
```

If `use_full_measure` is set, the `QTargetMeasure` will only ever contain a single `QTargetBeat` instance:

```
>>> another_q_target_measure = quantizationtools.QTargetMeasure(
...     offset_in_ms=1000,
...     search_tree=search_tree,
...     tempo=tempo,
...     time_signature=time_signature,
...     use_full_measure=True,
...     )
```

```
>>> for q_target_beat in another_q_target_measure.beats:
...     print q_target_beat.offset_in_ms, q_target_beat.duration_in_ms
1000 4000
```

Not composer-safe.

Used internally by `Quantizer`.

Return `QTargetMeasure` instance.

Read-only properties

`QTargetMeasure.`**`beats`**

The tuple of `QTargetBeats` contained by the `QTargetMeasure`:

```
>>> for q_target_beat in q_target_measure.beats:
...     q_target_beat
```

```
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(1000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={   2: None}
            ),
        tempo=contexttools.TempoMark(
            durationtools.Duration(1, 4),
            60
            )
        )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(2000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={   2: None}
            ),
        tempo=contexttools.TempoMark(
            durationtools.Duration(1, 4),
            60
            )
        )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(3000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={   2: None}
            ),
        tempo=contexttools.TempoMark(
            durationtools.Duration(1, 4),
            60
            )
        )
    quantizationtools.QTargetBeat(
        beatspan=durationtools.Duration(1, 4),
        offset_in_ms=durationtools.Offset(4000, 1),
        search_tree=quantizationtools.UnweightedSearchTree(
            definition={   2: None}
            ),
        tempo=contexttools.TempoMark(
            durationtools.Duration(1, 4),
            60
            )
        )
```

Return tuple.

`QTargetMeasure.`**`duration_in_ms`**

The duration in milliseconds of the `QTargetMeasure`:

```
>>> q_target_measure.duration_in_ms
Duration(4000, 1)
```

Return Duration.

`QTargetMeasure.`**`offset_in_ms`**

The offset in milliseconds of the `QTargetMeasure`:

```
>>> q_target_measure.offset_in_ms
Offset(1000, 1)
```

Return Offset.

`QTargetMeasure.`**`search_tree`**

The search tree of the `QTargetMeasure`:

```
>>> q_target_measure.search_tree
UnweightedSearchTree(
    definition={   2: None}
    )
```

Return `SearchTree` instance.

QTargetMeasure.**storage_format**
Storage format of Abjad object.

Return string.

QTargetMeasure.**tempo**
The tempo of the QTargetMeasure:

```
>>> q_target_measure.tempo
TempoMark(Duration(1, 4), 60)
```

Return TempoMark instance.

QTargetMeasure.**time_signature**
The time signature of the QTargetMeasure:

```
>>> q_target_measure.time_signature
TimeSignatureMark((4, 4))
```

Return 'TimeSignatureMark instance.

QTargetMeasure.**use_full_measure**
The use_full_measure flag of the QTargetMeasure:

```
>>> q_target_measure.use_full_measure
False
```

Return boolean.

Special methods

QTargetMeasure.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

QTargetMeasure.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QTargetMeasure.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

QTargetMeasure.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

QTargetMeasure.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

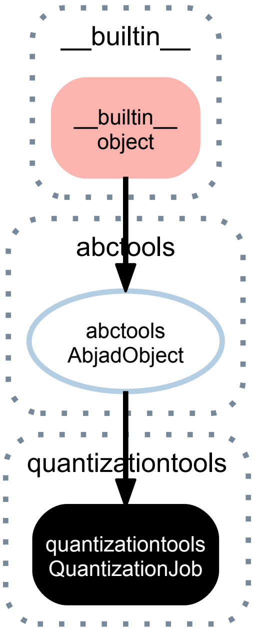
Raise exception.

QTargetMeasure.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

QTargetMeasure.**__repr__**()

22.2.24 quantizationtools.QuantizationJob



class quantizationtools.**QuantizationJob**(*job_id*, *search_tree*, *q_event_proxies*, *q_grids=None*)

A copiable, picklable class for generating all QGrids which are valid under a given SearchTree for a sequence of QEventProxies:

```
>>> q_event_a = quantizationtools.PitchedQEvent(250, [0, 1])
>>> q_event_b = quantizationtools.SilentQEvent(500)
>>> q_event_c = quantizationtools.PitchedQEvent(750, [3, 7])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.25)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.5)
>>> proxy_c = quantizationtools.QEventProxy(q_event_c, 0.75)
```

```
>>> definition = {2: {2: None}, 3: None, 5: None}
>>> search_tree = quantizationtools.UnweightedSearchTree(definition)
```

```
>>> job = quantizationtools.QuantizationJob(
...     1, search_tree, [proxy_a, proxy_b, proxy_c])
```

QuantizationJob generates QGrids when called, and stores those QGrids on its q_grids attribute, allowing them to be recalled later, even if pickled:

```
>>> job()
>>> for q_grid in job.q_grids:
...     print q_grid.rtm_format
1
(1 (1 1 1 1 1))
(1 (1 1 1))
(1 (1 1))
(1 ((1 (1 1)) (1 (1 1))))
```

QuantizationJob is intended to be useful in multiprocessing-enabled environments.

Return QuantizationJob instance.

Read-only properties

QuantizationJob.**job_id**

The job id of the QuantizationJob:

```
>>> job.job_id
1
```

Only meaningful when the job is processed via multiprocessing, as the job id is necessary to reconstruct the order of jobs.

Return int.

QuantizationJob.**q_event_proxies**
The QEventProxies the QuantizationJob was instantiated with:

```
>>> for q_event_proxy in job.q_event_proxies:
...     q_event_proxy
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(250, 1),
        (NamedChromaticPitch("c'"), NamedChromaticPitch("cs'")),
        attachments=()
        ),
    durationtools.Offset(1, 4)
    )
quantizationtools.QEventProxy(
    quantizationtools.SilentQEvent(
        durationtools.Offset(500, 1),
        attachments=()
        ),
    durationtools.Offset(1, 2)
    )
quantizationtools.QEventProxy(
    quantizationtools.PitchedQEvent(
        durationtools.Offset(750, 1),
        (NamedChromaticPitch("ef'"), NamedChromaticPitch("g'")),
        attachments=()
        ),
    durationtools.Offset(3, 4)
    )
```

Return tuple.

QuantizationJob.**q_grids**
The generated QGrids:

```
>>> for q_grid in job.q_grids:
...     print q_grid.rtm_format
1
(1 (1 1 1 1 1))
(1 (1 1 1))
(1 (1 1))
(1 ((1 (1 1)) (1 (1 1))))
```

Return tuple.

QuantizationJob.**search_tree**
The search tree the QuantizationJob was instantiated with:

```
>>> job.search_tree
UnweightedSearchTree(
    definition={   2: {   2: None}, 3: None, 5: None}
    )
```

Return SearchTree instance.

QuantizationJob.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

QuantizationJob.**__call__**()

`QuantizationJob.__eq__(expr)`

`QuantizationJob.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`QuantizationJob.__getstate__()`

`QuantizationJob.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`QuantizationJob.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`QuantizationJob.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`QuantizationJob.__ne__(expr)`
Defined equal to the opposite of equality.

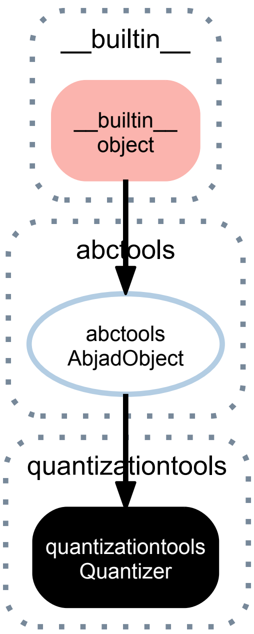
Return boolean.

`QuantizationJob.__repr__()`
Interpreter representation of Abjad object.

Return string.

`QuantizationJob.__setstate__(state)`

22.2.25 quantizationtools.Quantizer



class `quantizationtools.Quantizer`
`Quantizer` quantizes sequences of attack-points, encapsulated by `QEventSequences`, into score trees:

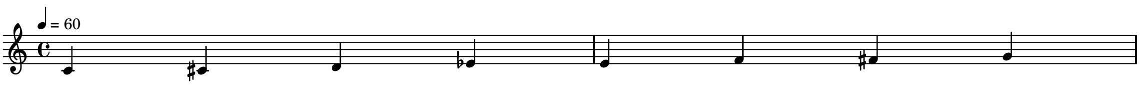
```
>>> quantizer = quantizationtools.Quantizer()
```

```
>>> durations = [1000] * 8
>>> pitches = range(8)
>>> q_event_sequence = quantizationtools.QEventSequence.from_millisecond_pitch_pairs(
...     zip(durations, pitches))
```

Quantization defaults to outputting into a 4/4, quarter=60 musical structure:

```
>>> result = quantizer(q_event_sequence)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            {
                \time 4/4
                \tempo 4=60
                c'4
                cs'4
                d'4
                ef'4
            }
            {
                e'4
                f'4
                fs'4
                g'4
            }
        }
    }
>>
```

```
>>> show(score)
```



However, the behavior of the `Quantizer` can be modified at call-time. Passing a `QSchema` instance will alter the macro-structure of the output.

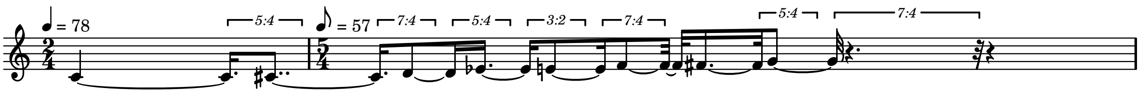
Here, we quantize using settings specified by a `MeasurewiseQSchema`, which will cause the `Quantizer` to group the output into measures with different tempi and time signatures:

```
>>> measurewise_q_schema = quantizationtools.MeasurewiseQSchema(
...     {'tempo': ((1, 4), 78), 'time_signature': (2, 4)},
...     {'tempo': ((1, 8), 57), 'time_signature': (5, 4)},
...     )
```

```
>>> result = quantizer(q_event_sequence, q_schema=measurewise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            {
                \time 2/4
                \tempo 4=78
                c'4 ~
                \times 4/5 {
                    c'16.
                    cs'8.. ~
                }
            }
            {
                \time 5/4
                \tempo 8=57
                \times 4/7 {
                    cs'16.
                    d'8 ~
                }
                \times 4/5 {
                    d'16
```

```
                    ef'16. ~
                }
                \times 2/3 {
                    ef'16
                    e'8 ~
                }
                \times 4/7 {
                    e'16
                    f'8 ~
                    f'32 ~
                }
                f'32
                fs'16. ~
                \times 4/5 {
                    fs'32
                    g'8 ~
                }
                \times 4/7 {
                    g'32
                    r4. ~
                    r32 ~
                }
                r4
            }
        }
    }
>>
```

```
>>> show(score)
```



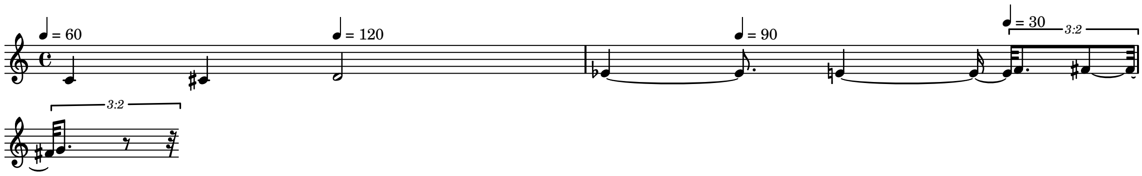
Here we quantize using settings specified by a `BeatwiseQSchema`, which keeps the output of the quantizer "flattened", without measures or explicit time signatures. The default beat-wise settings of quarter=60 persists until the third "beatspan":

```
>>> beatwise_q_schema = quantizationtools.BeatwiseQSchema(
... {
...     2: {'tempo': ((1, 4), 120)},
...     5: {'tempo': ((1, 4), 90)},
...     7: {'tempo': ((1, 4), 30)},
... })
```

```
>>> result = quantizer(q_event_sequence, q_schema=beatwise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            \tempo 4=60
            c'4
            cs'4
            \tempo 4=120
            d'2
            ef'4 ~
            \tempo 4=90
            ef'8.
            e'4 ~
            e'16 ~
            \times 2/3 {
                \tempo 4=30
                e'32
                f'8.
                fs'8 ~
                fs'32 ~
            }
            \times 2/3 {
                fs'32
                g'8.
                r8 ~
```

```
                        r32
                    }
                }
            }
        >>
```

```
>>> show(score)
```



Note that `TieChains` are generally fused together in the above example, but break at tempo changes.

Other keyword arguments are:

- `grace_handler`: a `GraceHandler` instance controls whether and how grace notes are used in the output. Options currently include `CollapsingGraceHandler`, `ConcatenatingGraceHandler` and `DiscardingGraceHandler`.
- `heuristic`: a `Heuristic` instance controls how output rhythms are selected from a pool of candidates. Options currently include the `DistanceHeuristic` class.
- `job_handler`: a `JobHandler` instance controls whether or not parallel processing is used during the quantization process. Options include the `SerialJobHandler` and `ParallelJobHandler` classes.
- `attack_point_optimizer`: an `AttackPointOptimizer` instance controls whether and how tie chains are re-notated. Options currently include `MeasurewiseAttackPointOptimizer`, `NaiveAttackPointOptimizer` and `NullAttackPointOptimizer`.

Refer to the reference pages for `BeatwiseQSchema` and `MeasurewiseQSchema` for more information on controlling the `Quantizer`'s output, and to the reference on `SearchTree` for information on controlling the rhythmic complexity of that same output.

Return `Quantizer` instance.

Read-only properties

`Quantizer.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`Quantizer.`**`__call__`**(*q_event_sequence*, *q_schema=None*, *grace_handler=None*, *heuristic=None*, *job_handler=None*, *attack_point_optimizer=None*, *attach_tempo_marks=True*)

`Quantizer.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`Quantizer.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Quantizer.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Quantizer.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Quantizer.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

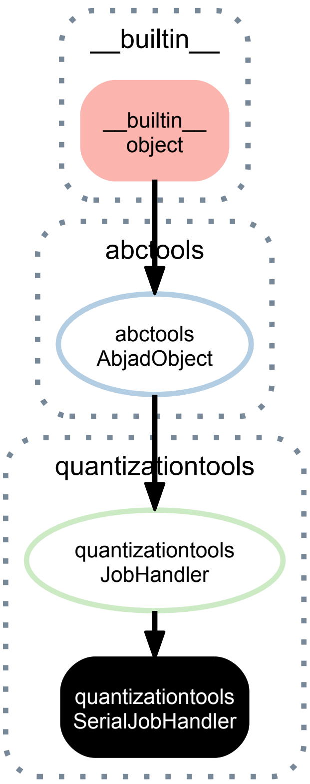
`Quantizer.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`Quantizer.__repr__()`
Interpreter representation of Abjad object.

Return string.

22.2.26 quantizationtools.SerialJobHandler



class `quantizationtools.SerialJobHandler`
Processes `QuantizationJob` instances sequentially.

Read-only properties

`SerialJobHandler.storage_format`
Storage format of Abjad object.

Return string.

Special methods

SerialJobHandler.**__call__**(*jobs*)

SerialJobHandler.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

SerialJobHandler.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SerialJobHandler.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SerialJobHandler.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SerialJobHandler.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

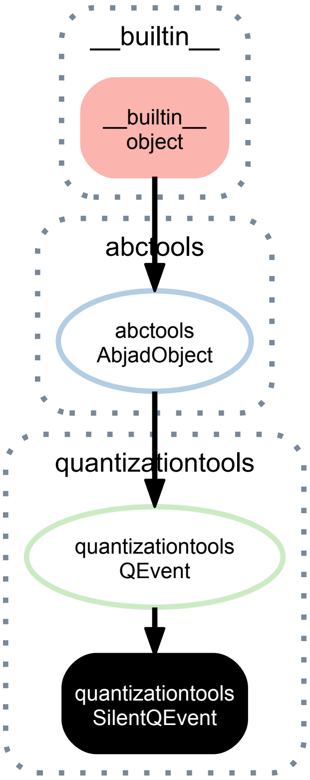
SerialJobHandler.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SerialJobHandler.**__repr__**()
Interpreter representation of Abjad object.

Return string.

22.2.27 quantizationtools.SilentQEvent



class `quantizationtools.`**`SilentQEvent`**(*offset*, *attachments=None*, *index=None*)

A `QEvent` which indicates the onset of a period of silence in a `QEventSequence`:

```
>>> q_event = quantizationtools.SilentQEvent(1000)
>>> q_event
quantizationtools.SilentQEvent(
    durationtools.Offset(1000, 1),
    attachments=()
    )
```

Return `SilentQEvent` instance.

Read-only properties

`SilentQEvent.`**`attachments`**

`SilentQEvent.`**`index`**

The optional index, for sorting QEvents with identical offsets.

`SilentQEvent.`**`offset`**

The offset in milliseconds of the event.

`SilentQEvent.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`SilentQEvent.`**`__eq__`**(*expr*)

`SilentQEvent.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SilentQEvent.**__getstate__**()

SilentQEvent.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SilentQEvent.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SilentQEvent.**__lt__**(*expr*)

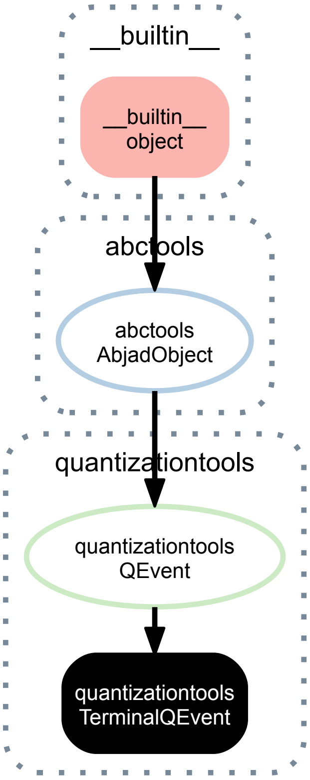
SilentQEvent.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SilentQEvent.**__repr__**()

SilentQEvent.**__setstate__**(*state*)

22.2.28 quantizationtools.TerminalQEvent



class quantizationtools.**TerminalQEvent**(*offset*)
The terminal event in a series of QEvents:

```
>>> q_event = quantizationtools.TerminalQEvent(1000)
>>> q_event
quantizationtools.TerminalQEvent(
    durationtools.Offset(1000, 1)
    )
```

Carries no significance outside the context of a QEventSequence.

Return TerminalQEvent instance.

Read-only properties

`TerminalQEvent.`**`index`**
The optional index, for sorting QEvents with identical offsets.

`TerminalQEvent.`**`offset`**
The offset in milliseconds of the event.

`TerminalQEvent.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`TerminalQEvent.`**`__eq__`**`(`*expr*`)`

`TerminalQEvent.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TerminalQEvent.`**`__getstate__`**`()`

`TerminalQEvent.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`TerminalQEvent.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TerminalQEvent.`**`__lt__`**`(`*expr*`)`

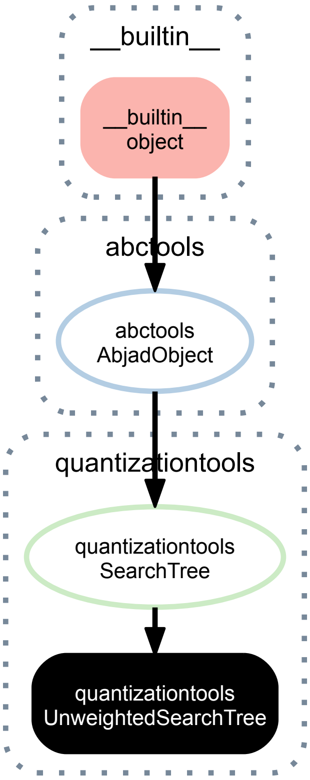
`TerminalQEvent.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`TerminalQEvent.`**`__repr__`**`()`

`TerminalQEvent.`**`__setstate__`**`(`*state*`)`

22.2.29 quantizationtools.UnweightedSearchTree



class `quantizationtools.`**`UnweightedSearchTree`**(*definition=None*)

Concrete `SearchTree` subclass, based on Paul Nauert's search tree model:

```
>>> search_tree = quantizationtools.UnweightedSearchTree()
>>> search_tree
UnweightedSearchTree(
    definition={    2: {    2: {    2: {    2: None}, 3: None}, 3: None, 5: None, 7: None},
        3: {    2: {    2: None}, 3: None, 5: None},
        5: {    2: None, 3: None},
        7: {    2: None},
        11: None,
        13: None}
    )
```

The search tree defines how nodes in a `QGrid` may be subdivided, if they happen to contain `QEvents` (or, in actuality, `QEventProxy` instances which reference `QEvents`, but rescale their offsets between `0` and `1`).

In the default definition, the root node of the `QGrid` may be subdivided into `2`, `3`, `5`, `7`, `11` or `13` equal parts. If divided into `2` parts, the divisions of the root node may be divided again into `2`, `3`, `5` or `7`, and so forth.

This definition is structured as a collection of nested dictionaries, whose keys are integers, and whose values are either the sentinel `None` indicating no further permissable divisions, or dictionaries obeying these same rules, which then indicate the possibilities for further division.

Calling a `UnweightedSearchTree` with a `QGrid` instance will generate all permissable subdivided `QGrids`, according to the definition of the called search tree:

```
>>> q_event_a = quantizationtools.PitchedQEvent(130, [0, 1, 4])
>>> q_event_b = quantizationtools.PitchedQEvent(150, [2, 3, 5])
>>> proxy_a = quantizationtools.QEventProxy(q_event_a, 0.5)
>>> proxy_b = quantizationtools.QEventProxy(q_event_b, 0.667)
>>> q_grid = quantizationtools.QGrid()
>>> q_grid.fit_q_events([proxy_a, proxy_b])
```

```
>>> q_grids = search_tree(q_grid)
>>> for grid in q_grids:
...     print grid.rtm_format
(1 (1 1))
(1 (1 1 1))
(1 (1 1 1 1 1))
(1 (1 1 1 1 1 1 1))
(1 (1 1 1 1 1 1 1 1 1 1 1))
(1 (1 1 1 1 1 1 1 1 1 1 1 1 1))
```

A custom `UnweightedSearchTree` may be defined by passing in a dictionary, as described above. The following search tree only permits divisions of the root node into `2s` and `3s`, and if divided into `2`, a node may be divided once more into `2` parts:

```
>>> definition = {2: {2: None}, 3: None}
>>> search_tree = quantizationtools.UnweightedSearchTree(definition)
```

```
>>> q_grids = search_tree(q_grid)
>>> for grid in q_grids:
...     print grid.rtm_format
(1 (1 1))
(1 (1 1 1))
```

Return `UnweightedSearchTree` instance.

Read-only properties

`UnweightedSearchTree.`**`default_definition`**
The default search tree definition, based on the search tree given by Paul Nauert:

```
>>> import pprint
>>> search_tree = quantizationtools.UnweightedSearchTree()
>>> pprint.pprint(search_tree.default_definition)
{2: {2: {2: {2: None}, 3: None}, 3: None, 5: None, 7: None},
 3: {2: {2: None}, 3: None, 5: None},
 5: {2: None, 3: None},
 7: {2: None},
 11: None,
 13: None}
```

Return dictionary.

`UnweightedSearchTree.`**`definition`**
The search tree definition.

Return dictionary.

`UnweightedSearchTree.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`UnweightedSearchTree.`**`__call__`**`(`*q_grid*`)`

`UnweightedSearchTree.`**`__eq__`**`(`*expr*`)`

`UnweightedSearchTree.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`UnweightedSearchTree.`**`__getstate__`**`()`

`UnweightedSearchTree.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`UnweightedSearchTree.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`UnweightedSearchTree.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

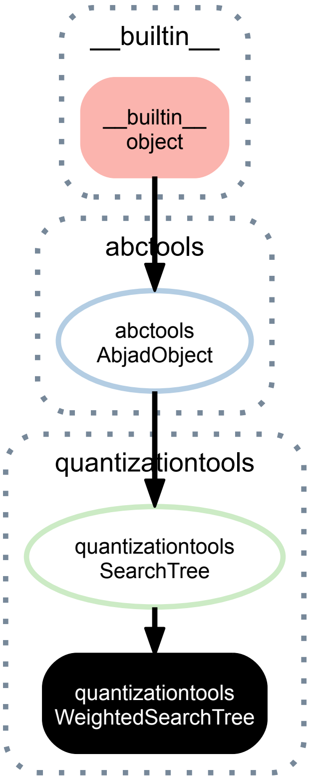
`UnweightedSearchTree.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`UnweightedSearchTree.__repr__()`

`UnweightedSearchTree.__setstate__(state)`

22.2.30 quantizationtools.WeightedSearchTree



class `quantizationtools.WeightedSearchTree(definition=None)`
A search tree that allows for dividing nodes in a `QGrid` into parts with unequal weights:

```
>>> search_tree = quantizationtools.WeightedSearchTree()
```

```
>>> search_tree
WeightedSearchTree(
    definition={   'divisors': (2, 3, 5, 7), 'max_depth': 3, 'max_divisions': 2}
    )
```

In `WeightedSearchTree`'s definition:

- `divisors` controls the sum of the parts of the ratio a node may be divided into,
- `max_depth` controls how many levels of tuplet nesting are permitted, and
- `max_divisions` controls the maximum permitted length of the weights in the ratio.

Thus, the default `WeightedSearchTree` permits the following ratios:

```
>>> for x in search_tree.all_compositions:
...     x
...
(1, 1)
(2, 1)
(1, 2)
(4, 1)
(3, 2)
(2, 3)
(1, 4)
(6, 1)
(5, 2)
(4, 3)
(3, 4)
(2, 5)
(1, 6)
```

Return `WeightedSearchTree` instance.

Read-only properties

`WeightedSearchTree.`**`all_compositions`**

`WeightedSearchTree.`**`default_definition`**

`WeightedSearchTree.`**`definition`**
The search tree definition.

Return dictionary.

`WeightedSearchTree.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`WeightedSearchTree.`**`__call__`**(*q_grid*)

`WeightedSearchTree.`**`__eq__`**(*expr*)

`WeightedSearchTree.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`WeightedSearchTree.`**`__getstate__`**()

`WeightedSearchTree.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`WeightedSearchTree.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`WeightedSearchTree.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

WeightedSearchTree.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

WeightedSearchTree.**__repr__**()

WeightedSearchTree.**__setstate__**(*state*)

22.3 Functions

22.3.1 quantizationtools.millisecond_pitch_pairs_to_q_events

quantizationtools.**millisecond_pitch_pairs_to_q_events**(*pairs*)
Convert a list of pairs of millisecond durations and pitches to a list of `QEvent` instances.

Pitch values must be one of the following:

1. A single chromatic pitch number, indicating a note,
2. None, indicating a silence, or
3. An iterable of chromatic pitch numbers, indicating a chord.

```
>>> durations = [1001, 503, 230, 1340]
>>> pitches = [None, 0, (1, 2, 3), 4.5]
>>> pairs = zip(durations, pitches)
>>> for x in quantizationtools.millisecond_pitch_pairs_to_q_events(pairs): x
...
quantizationtools.SilentQEvent(
    durationtools.Offset(0, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1001, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1504, 1),
    (NamedChromaticPitch("cs'"), NamedChromaticPitch("d'"), NamedChromaticPitch("ef'")),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(1734, 1),
    (NamedChromaticPitch("eqs'"),),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(3074, 1)
    )
```

Return a list of `QEvent` instances.

22.3.2 quantizationtools.milliseconds_to_q_events

quantizationtools.**milliseconds_to_q_events**(*milliseconds*, *fuse_silences=False*)
Convert a list of millisecond durations to a list of `QEvent` instances.

Negative duration values can be used to indicate silence. Any resulting pitched `QEvent` instances will default to using middle-C.

```
>>> durations = [100, -250, 500]
>>> for x in quantizationtools.milliseconds_to_q_events(durations): x
...
quantizationtools.PitchedQEvent(
```

```
        durationtools.Offset(0, 1),
        (NamedChromaticPitch("c'"),),
        attachments=()
        )
    quantizationtools.SilentQEvent(
        durationtools.Offset(100, 1),
        attachments=()
        )
    quantizationtools.PitchedQEvent(
        durationtools.Offset(350, 1),
        (NamedChromaticPitch("c'"),),
        attachments=()
        )
    quantizationtools.TerminalQEvent(
        durationtools.Offset(850, 1)
        )
```

Return a list of `QEvent` instances.

22.3.3 quantizationtools.tempo_scaled_duration_to_milliseconds

`quantizationtools.`**`tempo_scaled_duration_to_milliseconds`**(*duration*, *tempo*)

Return the millisecond value of *duration* at *tempo*.

```
>>> duration = (1, 4)
>>> tempo = contexttools.TempoMark((1, 4), 60)
>>> quantizationtools.tempo_scaled_duration_to_milliseconds(
...     duration, tempo)
Duration(1000, 1)
```

Return `Duration` instance.

22.3.4 quantizationtools.tempo_scaled_durations_to_q_events

`quantizationtools.`**`tempo_scaled_durations_to_q_events`**(*durations*, *tempo*)

Convert a list of rational durations to a list of `QEvent` instances.

Negative duration values can be used to indicate silence. Any resulting pitched `QEvents` will default to using middle-C.

```
>>> durations = [Duration(-1, 2), Duration(1, 4), Duration(1, 6)]
>>> tempo = contexttools.TempoMark((1, 4), 55)
>>> for x in quantizationtools.tempo_scaled_durations_to_q_events(
...     durations, tempo): x
...
quantizationtools.SilentQEvent(
    durationtools.Offset(0, 1),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(24000, 11),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(36000, 11),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(4000, 1)
    )
```

Return a list of `QEvent` instances.

22.3.5 quantizationtools.tempo_scaled_leaves_to_q_events

quantizationtools.**tempo_scaled_leaves_to_q_events**(*leaves*, *tempo=None*)

Convert *leaves* to a list of QEvent objects. If the leaves have no effective tempo, *tempo* must be a TempoMark.

```
>>> source = Staff("c'4 r4. e'8 <g' b' d''>2")
>>> source_tempo = contexttools.TempoMark((1, 4), 55)
>>> for x in quantizationtools.tempo_scaled_leaves_to_q_events(
...     source[:], tempo=source_tempo): x
...
quantizationtools.PitchedQEvent(
    durationtools.Offset(0, 1),
    (NamedChromaticPitch("c'"),),
    attachments=()
    )
quantizationtools.SilentQEvent(
    durationtools.Offset(12000, 11),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(30000, 11),
    (NamedChromaticPitch("e'"),),
    attachments=()
    )
quantizationtools.PitchedQEvent(
    durationtools.Offset(36000, 11),
    (NamedChromaticPitch("g'"), NamedChromaticPitch("b'"), NamedChromaticPitch("d''")),
    attachments=()
    )
quantizationtools.TerminalQEvent(
    durationtools.Offset(60000, 11)
    )
```

Return a list of QEvent objects.

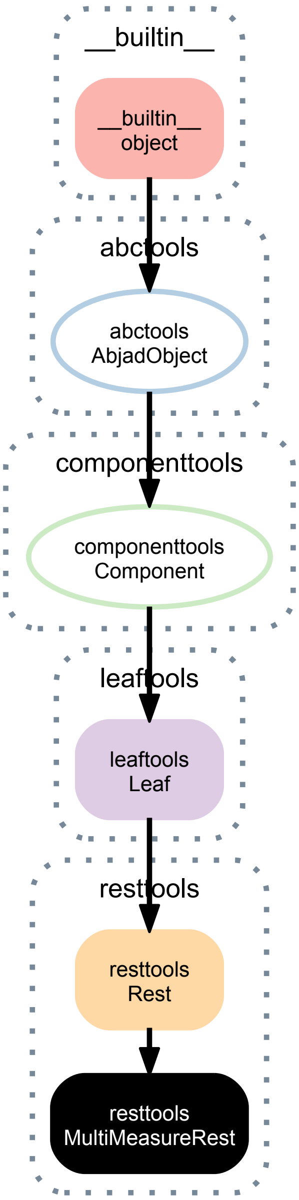
CHAPTER

TWENTYTHREE

RESTTOOLS

23.1 Concrete Classes

23.1.1 resttools.MultiMeasureRest



class resttools.**MultiMeasureRest**(**args*, ***kwargs*)

New in version 2.0. Abjad model of a multi-measure rest:

```
>>> resttools.MultiMeasureRest((1, 4))
MultiMeasureRest('R4')
```

Multi-measure rests are immutable.

Read-only properties

MultiMeasureRest.**descendants**
Read-only reference to component descendants score selection.

MultiMeasureRest.**duration**

MultiMeasureRest.**duration_in_seconds**

MultiMeasureRest.**leaf_index**

MultiMeasureRest.**lilypond_format**

MultiMeasureRest.**lineage**
Read-only reference to component lineage score selection.

MultiMeasureRest.**multiplied_duration**

MultiMeasureRest.**override**
Read-only reference to LilyPond grob override component plug-in.

MultiMeasureRest.**parent**

MultiMeasureRest.**parentage**
Read-only reference to component parentage score selection.

MultiMeasureRest.**preprolated_duration**

MultiMeasureRest.**prolation**

MultiMeasureRest.**set**
Read-only reference LilyPond context setting component plug-in.

MultiMeasureRest.**spanners**
Read-only reference to unordered set of spanners attached to component.

MultiMeasureRest.**storage_format**
Storage format of Abjad object.

Return string.

MultiMeasureRest.**timespan**
Read-only timespan of component.

MultiMeasureRest.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

MultiMeasureRest.**duration_multiplier**

MultiMeasureRest.**written_duration**

MultiMeasureRest.**written_pitch_indication_is_at_sounding_pitch**

MultiMeasureRest.**written_pitch_indication_is_nonsemantic**

Special methods

MultiMeasureRest.**__and__**(*arg*)

MultiMeasureRest.**__copy__**(**args*)

MultiMeasureRest.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

MultiMeasureRest.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MultiMeasureRest.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MultiMeasureRest.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MultiMeasureRest.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MultiMeasureRest.**__mul__**(*n*)

MultiMeasureRest.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MultiMeasureRest.**__or__**(*arg*)

MultiMeasureRest.**__repr__**()

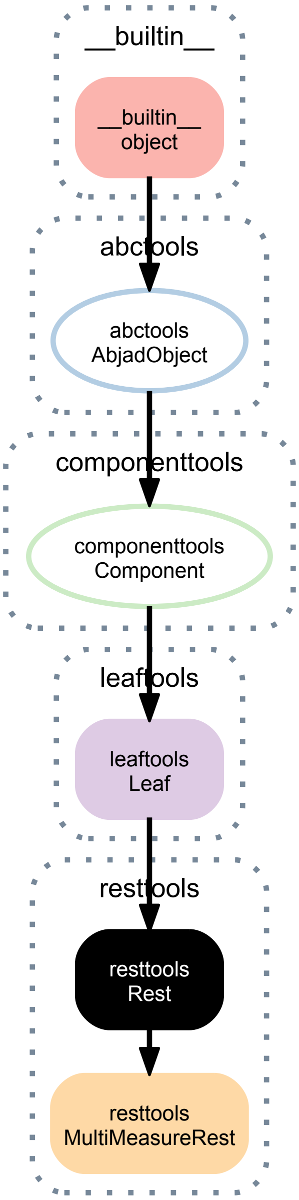
MultiMeasureRest.**__rmul__**(*n*)

MultiMeasureRest.**__str__**()

MultiMeasureRest.**__sub__**(*arg*)

MultiMeasureRest.**__xor__**(*arg*)

23.1.2 resttools.Rest



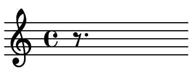
class resttools.**Rest** (**args*, ***kwargs*)

Abjad model of a rest:

```
>>> rest = Rest((3, 16))
```

```
>>> rest
Rest('r8.')
```

```
>>> show(rest)
```



Return Rest instance.

Read-only properties

Rest.**descendants**

Read-only reference to component descendants score selection.

Rest.**duration**

Rest.**duration_in_seconds**

Rest.**leaf_index**

Rest.**lilypond_format**

Rest.**lineage**
Read-only reference to component lineage score selection.

Rest.**multiplied_duration**

Rest.**override**
Read-only reference to LilyPond grob override component plug-in.

Rest.**parent**

Rest.**parentage**
Read-only reference to component parentage score selection.

Rest.**preprolated_duration**

Rest.**prolation**

Rest.**set**
Read-only reference LilyPond context setting component plug-in.

Rest.**spanners**
Read-only reference to unordered set of spanners attached to component.

Rest.**storage_format**
Storage format of Abjad object.

Return string.

Rest.**timespan**
Read-only timespan of component.

Rest.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

Rest.**duration_multiplier**

Rest.**written_duration**

Rest.**written_pitch_indication_is_at_sounding_pitch**

Rest.**written_pitch_indication_is_nonsemantic**

Special methods

Rest.**__and__**(*arg*)

Rest.**__copy__**(**args*)

Rest.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Rest.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Rest.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Rest.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Rest.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Rest.**__mul__**(*n*)

Rest.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Rest.**__or__**(*arg*)

Rest.**__repr__**()

Rest.**__rmul__**(*n*)

Rest.**__str__**()

Rest.**__sub__**(*arg*)

Rest.**__xor__**(*arg*)

23.2 Functions

23.2.1 resttools.all_are_rests

resttools.**all_are_rests**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad rests:

```
>>> rests = [Rest('r4'), Rest('r4'), Rest('r4')]
```

```
>>> resttools.all_are_rests(rests)
True
```

True when *expr* is an empty sequence:

```
>>> resttools.all_are_rests([])
True
```

Otherwise false:

```
>>> resttools.all_are_rests('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

23.2.2 resttools.make_multi_measure_rests

resttools.**make_multi_measure_rests**(*durations*)
New in version 2.0. Make multi-measure rests from *durations*:

```
>>> resttools.make_multi_measure_rests([(4, 4), (7, 4)])
[MultiMeasureRest('R1'), MultiMeasureRest('R1..')]
```

Return list.

23.2.3 resttools.make_repeated_rests_from_time_signature

resttools.**make_repeated_rests_from_time_signature**(*time_signature*)

New in version 2.0. Make repeated rests from *time_signature*:

```
>>> resttools.make_repeated_rests_from_time_signature((5, 32))
[Rest('r32'), Rest('r32'), Rest('r32'), Rest('r32'), Rest('r32')]
```

Return list of newly constructed rests.

23.2.4 resttools.make_repeated_rests_from_time_signatures

resttools.**make_repeated_rests_from_time_signatures**(*time_signatures*)

Make repated rests from *time_signatures*:

```
resttools.make_repeated_rests_from_time_signatures([(2, 8), (3, 32)])
[[Rest('r8'), Rest('r8')], [Rest('r32'), Rest('r32'), Rest('r32')]]
```

Return two-dimensional list of newly constructed rest lists.

Use `sequencetools.flatten_sequence()` to flatten output if required.

23.2.5 resttools.make_rests

resttools.**make_rests**(*durations*, *decrease_durations_monotonically=True*, *tie_parts=False*)

New in version 1.1. Make rests.

Make rests and drecrease durations monotonically:

```
>>> resttools.make_rests([(5, 16), (9, 16)], decrease_durations_monotonically=True)
[Rest('r4'), Rest('r16'), Rest('r2'), Rest('r16')]
```

Makes rests and increase durations monotonically:

```
>>> resttools.make_rests([(5, 16), (9, 16)], decrease_durations_monotonically=False)
[Rest('r16'), Rest('r4'), Rest('r16'), Rest('r2')]
```

Make tied rests:

```
>>> voice = Voice(resttools.make_rests([(5, 16), (9, 16)], tie_parts=True))
```

```
>>> f(voice)
\new Voice {
    r4 ~
    r16
    r2 ~
    r16
}
```

Return list of rests.

23.2.6 resttools.make_tied_rest

resttools.**make_tied_rest**(*duration*, *decrease_durations_monotonically=True*, *forbidden_written_duration=None*, *tie_parts=False*)

Returns a list of rests to fill given duration.

Tie rest parts when `tie_parts=True`.

23.2.7 resttools.replace_leaves_in_expr_with_rests

resttools.**replace_leaves_in_expr_with_rests**(*expr*)
New in version 1.1. Replace leaves in *expr* with rests:

```
>>> staff = Staff(Measure((2, 8), "c'8 d'8") * 2)
>>> resttools.replace_leaves_in_expr_with_rests(staff[0])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        r8
        r8
    }
    {
        c'8
        d'8
    }
}
```

Return none.

23.2.8 resttools.set_vertical_positioning_pitch_on_rest

resttools.**set_vertical_positioning_pitch_on_rest**(*rest*, *pitch*)
New in version 2.0. Set vertical positioning *pitch* on *rest*:

```
>>> rest = Rest((1, 4))
```

```
>>> resttools.set_vertical_positioning_pitch_on_rest(rest, "d''")
Rest('r4')
```

```
>>> f(rest)
d''4 \rest
```

Raise type error when *rest* is not a rest.

Return *rest*.

23.2.9 resttools.yield_groups_of_rests_in_sequence

resttools.**yield_groups_of_rests_in_sequence**(*sequence*)
New in version 2.0. Yield groups of rests in *sequence*:

```
>>> staff = Staff("c'8 d'8 r8 r8 <e' g'>8 <f' a'>8 g'8 a'8 r8 r8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    r8
    r8
    <e' g'>8
    <f' a'>8
    g'8
    a'8
    r8
    r8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for rest in resttools.yield_groups_of_rests_in_sequence(staff):
...     rest
...
```

```
(Rest('r8'), Rest('r8'))
(Rest('r8'), Rest('r8'))
```

Return generator.

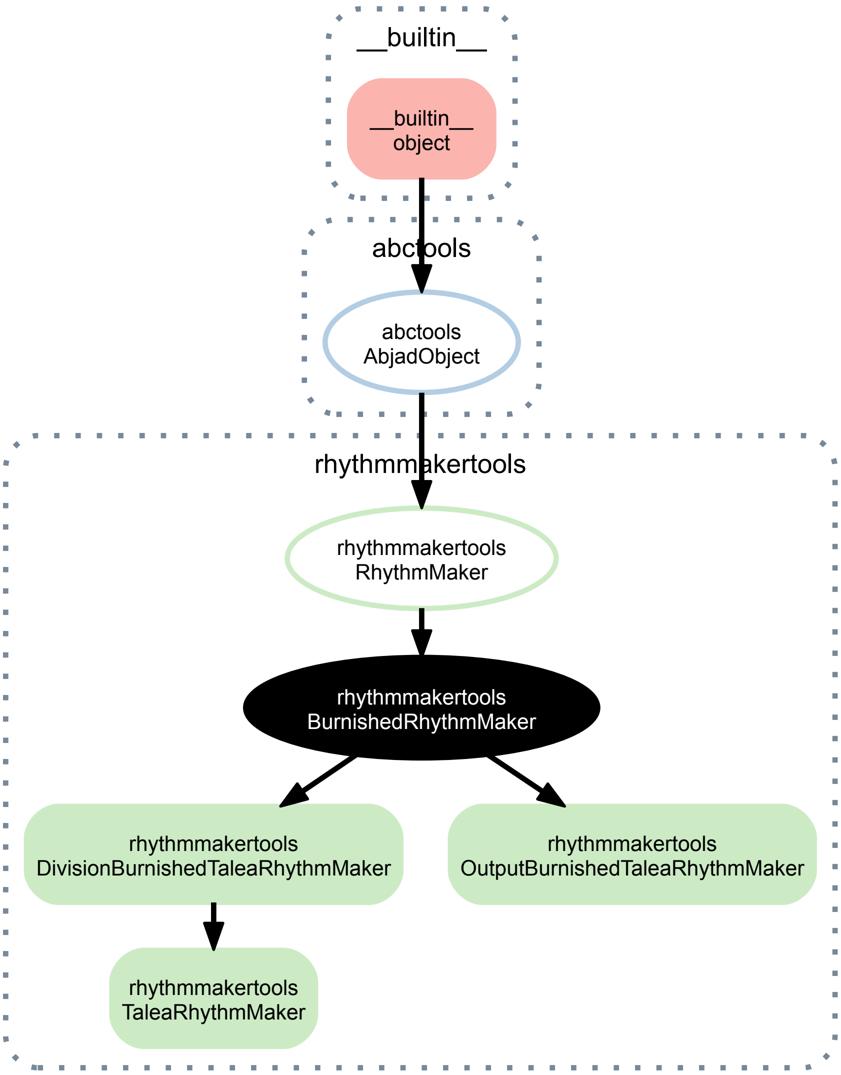
CHAPTER

TWENTYFOUR

RHYTHMMAKERTOOLS

24.1 Abstract Classes

24.1.1 rhythmmakertools.BurnishedRhythmMaker



class rhythmmakertools.**BurnishedRhythmMaker**(*talea=None, talea_denominator=None, prolation_addenda=None, lefts=None, middles=None, rights=None, left_lengths=None, right_lengths=None, secondary_divisions=None, talea_helper=None, prolation_addenda_helper=None, lefts_helper=None, middles_helper=None, rights_helper=None, left_lengths_helper=None, right_lengths_helper=None, secondary_divisions_helper=None, beam_each_cell=False, beam_cells_together=False, decrease_durations_monotonically=True, tie_split_notes=False, tie_rests=False*)

New in version 2.8. Abstract base class for rhythm-makers that burnish some or all of the output cells they produce.

'Burnishing' means to forcibly cast the first or last (or both first and last) elements of a output cell to be either a note or rest.

'Division-burnishing' rhythm-makers burnish every output cell they produce.

'Output-burnishing' rhythm-makers burnish only the first and last output cells they produce and leave interior output cells unchanged.

Read-only properties

BurnishedRhythmMaker.**storage_format**

Burnished rhythm-maker storage format.

Return string.

Methods

BurnishedRhythmMaker.**new**(***kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker.new(decrease_durations_monotonically=False)(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/16
        c'16 ~
        c'4
    }
    {
        \time 3/8
        c'4.
    }
}
```

Return new rhythm-maker.

`BurnishedRhythmMaker.`**`reverse`**`()`

Reverse burnished rhythm-maker.

Defined equal to a copy of rhythm-maker with all the following lists reversed:

```
new.talea
new.prolation_addenda
new.lefts
new.middles
new.rights
new.left_lengths
new.right_lengths
new.secondary_divisions
```

Return newly constructed rhythm-maker.

Special methods

`BurnishedRhythmMaker.`**`__call__`**`(`*divisions*, *seeds=None*`)`

Call burnished rhythm-maker on *divisions*.

Return either list of tuplets or else list of note-lists.

`BurnishedRhythmMaker.`**`__eq__`**`(`*expr*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`BurnishedRhythmMaker.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`BurnishedRhythmMaker.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`BurnishedRhythmMaker.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`BurnishedRhythmMaker.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`BurnishedRhythmMaker.`**`__ne__`**`(`*expr*`)`

Defined equal to the opposite of equality.

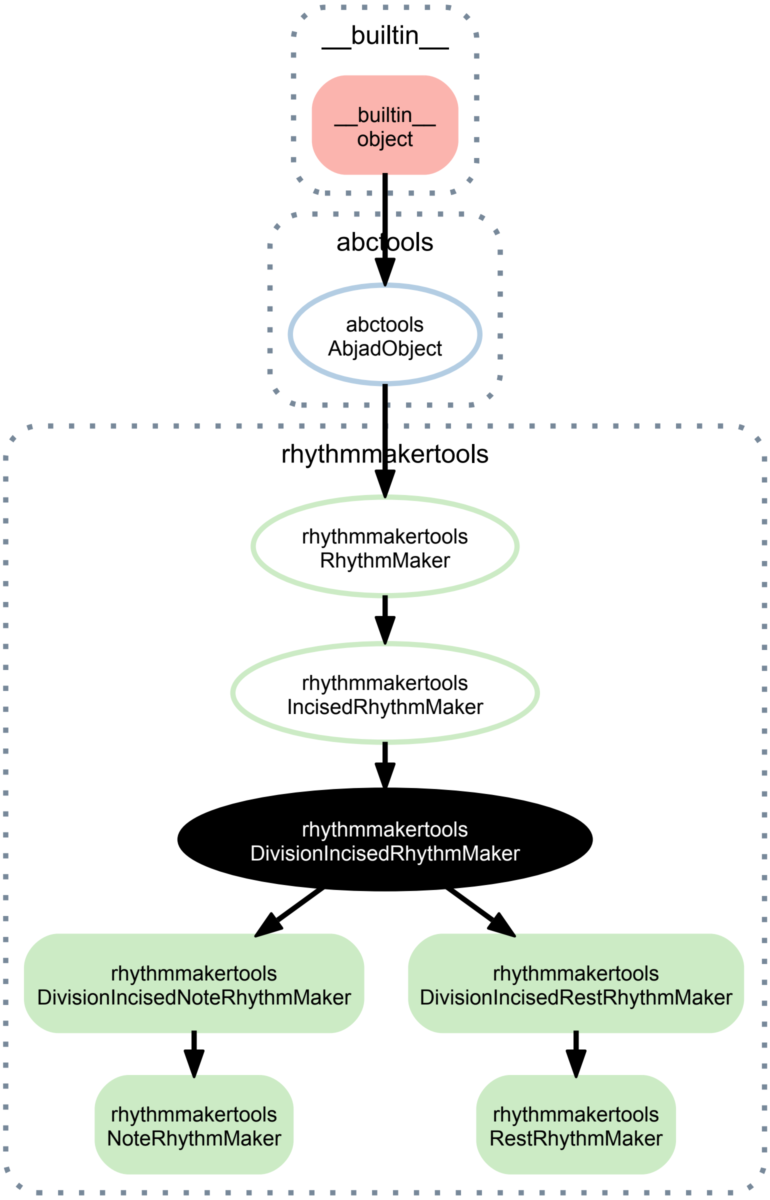
Return boolean.

`BurnishedRhythmMaker.`**`__repr__`**`()`

Rhythm-maker interpreter representation.

Return string.

24.1.2 rhythmmakertools.DivisionIncisedRhythmMaker



class rhythmmakertools.**DivisionIncisedRhythmMaker** (*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

New in version 2.8. Abstract base class for rhythm-makers that incise every output cell they produce.

Read-only properties

DivisionIncisedRhythmMaker.**storage_format**

Storage format of Abjad object.

Return string.

Methods

DivisionIncisedRhythmMaker.**new** (***kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker.new(decrease_durations_monotonically=False)(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/16
        c'16 ~
        c'4
    }
    {
        \time 3/8
        c'4.
    }
}
```

Return new rhythm-maker.

DivisionIncisedRhythmMaker.**reverse** ()

Reverse incised rhythm-maker.

Return newly constructed rhythm-maker.

Special methods

`DivisionIncisedRhythmMaker.__call__(divisions, seeds=None)`
Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`DivisionIncisedRhythmMaker.__eq__(expr)`
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`DivisionIncisedRhythmMaker.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedRhythmMaker.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`DivisionIncisedRhythmMaker.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedRhythmMaker.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

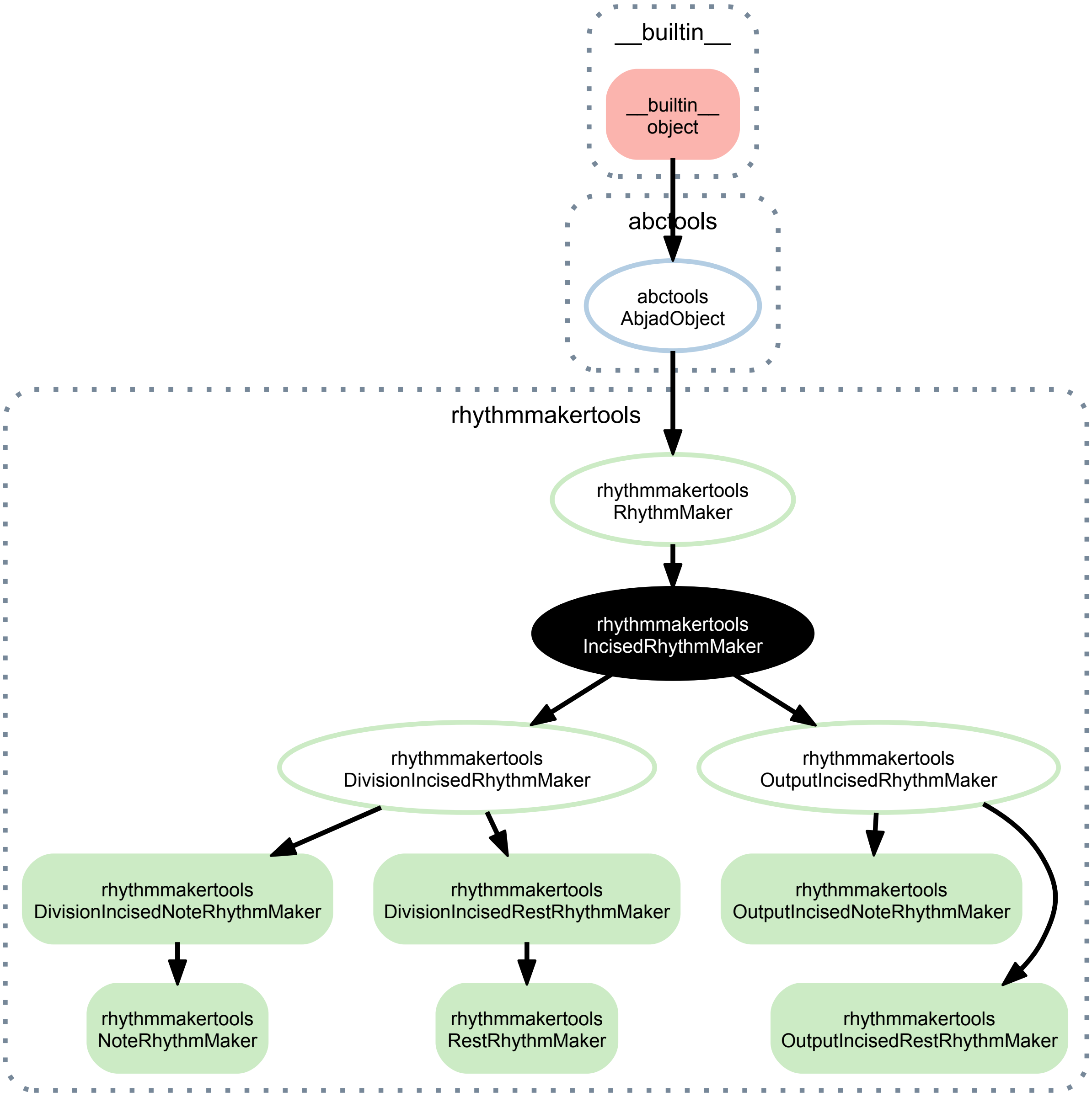
`DivisionIncisedRhythmMaker.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`DivisionIncisedRhythmMaker.__repr__()`
Rhythm-maker interpreter representation.

Return string.

24.1.3 rhythmmakertools.IncisedRhythmMaker



class `rhythmmakertools.IncisedRhythmMaker`(*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

Abstract base class for rhythm-makers that incise some or all of the output cells they produce.

Rhythm makers can incise the edge of every output cell.

Or rhythm-makers can incise only the start of the first output cell and the end of the last output cell.

Read-only properties

`IncisedRhythmMaker.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`IncisedRhythmMaker.`**`new`**`(`*`**kwargs`*`)`

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker.new(decrease_durations_monotonically=False)(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/16
        c'16 ~
        c'4
    }
    {
        \time 3/8
        c'4.
    }
}
```

Return new rhythm-maker.

`IncisedRhythmMaker.`**`reverse`**`()`

Reverse incised rhythm-maker.

Return newly constructed rhythm-maker.

Special methods

`IncisedRhythmMaker.`**`__call__`**`(`*`divisions, seeds=None`*`)`

Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`IncisedRhythmMaker.`**`__eq__`**`(`*`expr`*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`IncisedRhythmMaker.`**`__ge__`**`(`*`expr`*`)`

Abjad objects by default do not implement this method.

Raise exception.

`IncisedRhythmMaker.`**`__gt__`**`(`*`expr`*`)`

Abjad objects by default do not implement this method.

Raise exception

IncisedRhythmMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

IncisedRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

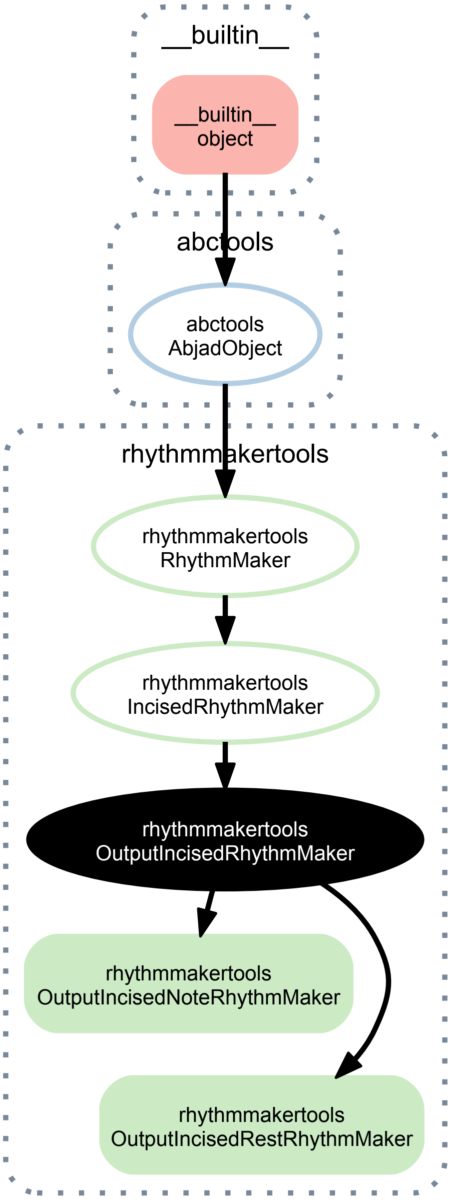
IncisedRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

IncisedRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

24.1.4 rhythmmakertools.OutputIncisedRhythmMaker



class rhythmmakertools.**OutputIncisedRhythmMaker**(*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

New in version 2.8. Abstract base class for rhythm-makers that incise only the first and last output cells they produce.

Read-only properties

OutputIncisedRhythmMaker.**storage_format**

Storage format of Abjad object.

Return string.

Methods

OutputIncisedRhythmMaker.**new**(***kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker.new(decrease_durations_monotonically=False)(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/16
        c'16 ~
        c'4
    }
    {
        \time 3/8
        c'4.
    }
}
```

Return new rhythm-maker.

OutputIncisedRhythmMaker.**reverse**()

Reverse incised rhythm-maker.

Return newly constructed rhythm-maker.

Special methods

`OutputIncisedRhythmMaker.__call__`(*divisions*, *seeds=None*)
Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`OutputIncisedRhythmMaker.__eq__`(*expr*)
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`OutputIncisedRhythmMaker.__ge__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OutputIncisedRhythmMaker.__gt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`OutputIncisedRhythmMaker.__le__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OutputIncisedRhythmMaker.__lt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

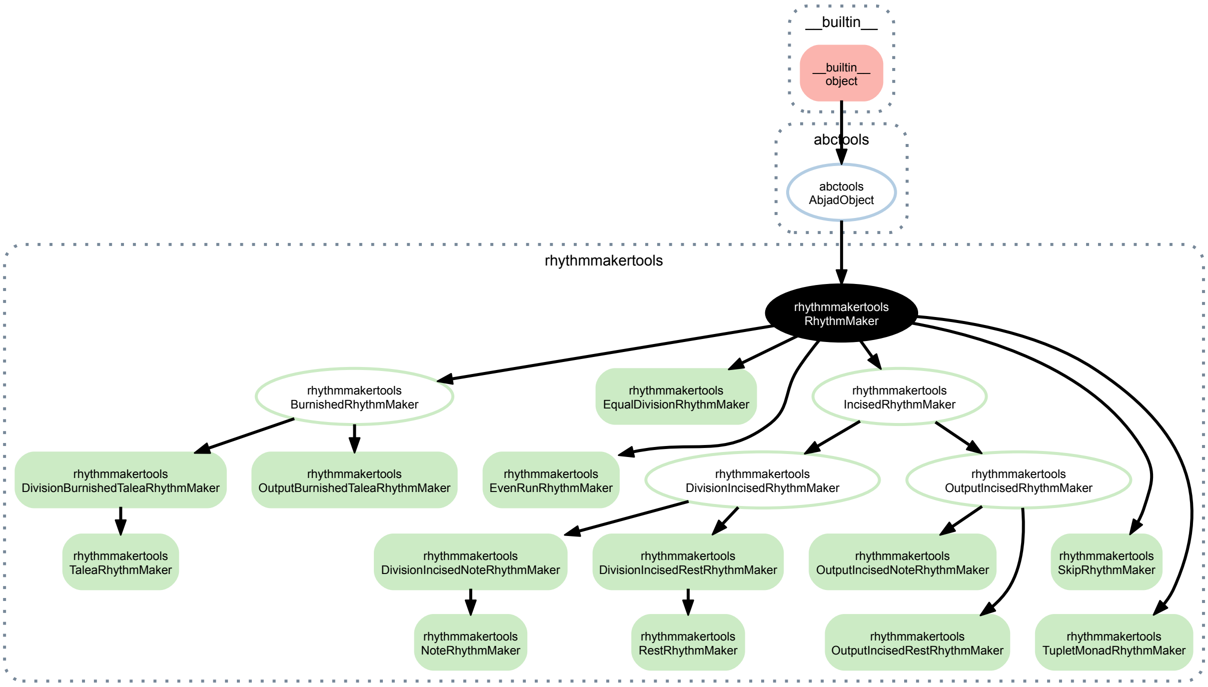
`OutputIncisedRhythmMaker.__ne__`(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`OutputIncisedRhythmMaker.__repr__`()
Rhythm-maker interpreter representation.

Return string.

24.1.5 rhythmmakertools.RhythmMaker



class `rhythmmakertools.`**`RhythmMaker`**(*forbidden_written_duration=None*, *beam_each_cell=True*, *beam_cells_together=False*)

New in version 2.8. Rhythm maker abstract base class.

Read-only properties

`RhythmMaker.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`RhythmMaker.`**`new`**(***kwargs*)

Create new rhythm-maker with *kwargs*:

```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker.new(decrease_durations_monotonically=False)(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(measuretools.make_measures_with_full_measure_spacer_skips(divisions))
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/16
        c'16 ~
        c'4
    }
    {
        \time 3/8
        c'4.
    }
}
```

Return new rhythm-maker.

`RhythmMaker.`**`reverse`**`()`

Reverse rhythm-maker.

Note: method is provisional.

Defined equal to exact copy of rhythm-maker.

This is the fallback for child classes.

Directed rhythm-maker child classes should override this method.

Return newly constructed rhythm-maker.

Special methods

`RhythmMaker.`**`__call__`**`(`*divisions*, *seeds=None*`)`

Cast *divisions* into duration pairs. Reduce numerator and denominator relative to each other.

Change none *seeds* into empty list.

Return duration pairs and seed list.

`RhythmMaker.`**`__eq__`**`(`*expr*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`RhythmMaker.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`RhythmMaker.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`RhythmMaker.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`RhythmMaker.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`RhythmMaker.`**`__ne__`**`(`*expr*`)`

Defined equal to the opposite of equality.

Return boolean.

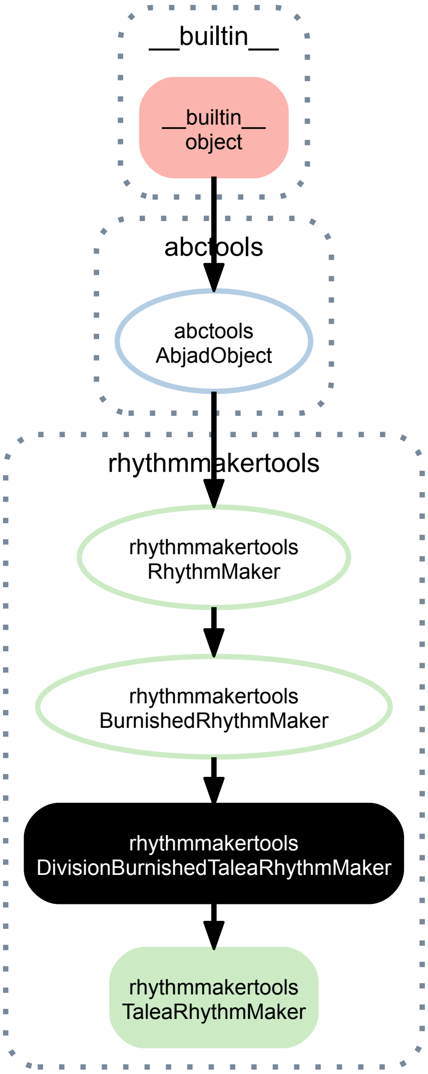
`RhythmMaker.`**`__repr__`**`()`

Rhythm-maker interpreter representation.

Return string.

24.2 Concrete Classes

24.2.1 rhythmmakertools.DivisionBurnishedTaleaRhythmMaker



class `rhythmmakertools.`**`DivisionBurnishedTaleaRhythmMaker`**(*talea=None, talea_denominator=None, prolation_addenda=None, lefts=None, middles=None, rights=None, left_lengths=None, right_lengths=None, secondary_divisions=None, talea_helper=None, prolation_addenda_helper=None, lefts_helper=None, middles_helper=None, rights_helper=None, left_lengths_helper=None, right_lengths_helper=None, secondary_divisions_helper=None, beam_each_cell=False, beam_cells_together=False, decrease_durations_monotonically=True, tie_split_notes=False, tie_rests=False*)

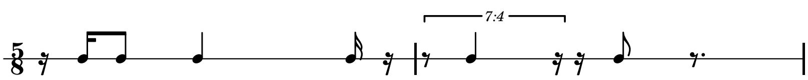
New in version 2.8. Division-burnished talea rhythm-maker:

```
>>> maker = rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
...     talea=[1, 1, 2, 4],
...     talea_denominator=16,
...     prolation_addenda=[0, 3],
...     lefts=[-1],
...     middles=[0],
...     rights=[-1],
...     left_lengths=[1],
...     right_lengths=[1],
...     secondary_divisions=[14])
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = [(5, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

`DivisionBurnishedTaleaRhythmMaker.`**`storage_format`**

Division-burnished talea rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
    )
```

Return string.

Methods

DivisionBurnishedTaleaRhythmMaker.**new**(***kwargs*)

Create new rhythm-maker with *kwargs*:

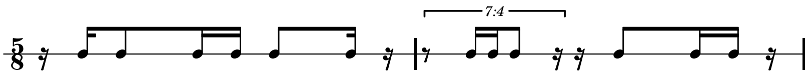
```
>>> z(maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
    )
```

```
>>> new_maker = maker.new(talea=[1, 1, 2])
```

```
>>> z(new_maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
    )
```

```
>>> divisions = [(5, 8), (5, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new division-burnished talea rhythm-maker.

`DivisionBurnishedTaleaRhythmMaker.reverse()`

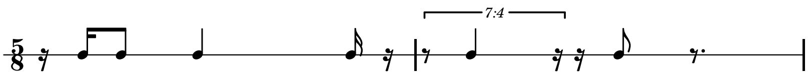
Reverse division-burnished talea rhythm-maker.

Nonreversed output:

```
>>> z(maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[1, 1, 2, 4],
    talea_denominator=16,
    prolation_addenda=[0, 3],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
    )
```

```
>>> divisions = [(5, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



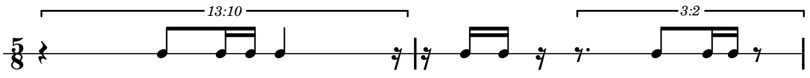
Reversed output:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker(
    talea=[4, 2, 1, 1],
    talea_denominator=16,
    prolation_addenda=[3, 0],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[14],
    beam_each_cell=False,
    beam_cells_together=False,
    decrease_durations_monotonically=False,
    tie_split_notes=False,
    tie_rests=False
    )
```

```
>>> divisions = [(5, 8), (5, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new division-burnished talea rhythm-maker.

Special methods

DivisionBurnishedTaleaRhythmMaker.**__call__**(*divisions*, *seeds=None*)
Call burnished rhythm-maker on *divisions*.

Return either list of tuplets or else list of note-lists.

DivisionBurnishedTaleaRhythmMaker.**__eq__**(*expr*)
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

DivisionBurnishedTaleaRhythmMaker.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DivisionBurnishedTaleaRhythmMaker.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DivisionBurnishedTaleaRhythmMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DivisionBurnishedTaleaRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

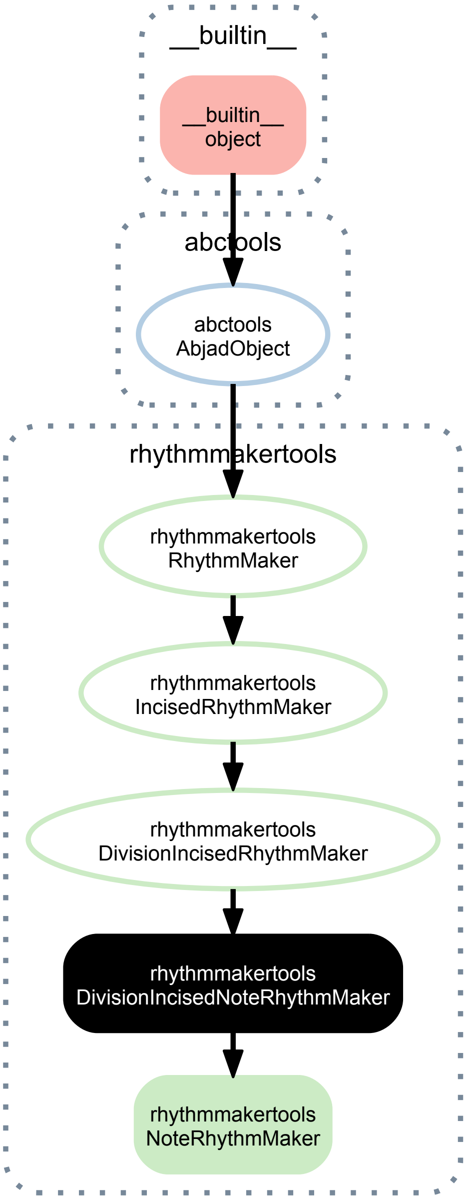
DivisionBurnishedTaleaRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DivisionBurnishedTaleaRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

24.2.2 rhythmmakertools.DivisionIncisedNoteRhythmMaker



class rhythmmakertools.**DivisionIncisedNoteRhythmMaker**(*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

New in version 2.8. Division-incised note rhythm-maker:

Example 1. Basic usage:

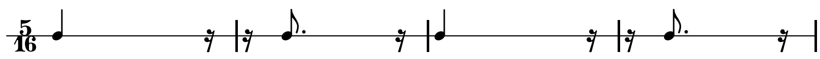
```
>>> maker = rhythmmakertools.DivisionIncisedNoteRhythmMaker(
...     prefix_talea=[-1],
...     prefix_lengths=[0, 1],
...     suffix_talea=[-1],
...     suffix_lengths=[1],
...     talea_denominator=16)
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = 4 * [(5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new RhythmicStaff {
    {
        \time 5/16
        c'4
        r16
    }
    {
        r16
        c'8.
        r16
    }
    {
        c'4
        r16
    }
    {
        r16
        c'8.
        r16
    }
}
```

```
>>> show(staff)
```



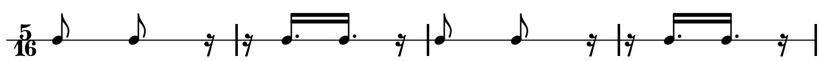
Example 2. Set *body_ratio* to divide middle part proportionally:

```
>>> maker = rhythmmakertools.DivisionIncisedNoteRhythmMaker(
...     prefix_talea=[-1],
...     prefix_lengths=[0, 1],
...     suffix_talea=[-1],
...     suffix_lengths=[1],
...     talea_denominator=16,
...     body_ratio=(1, 1))
```

```
>>> divisions = 4 * [(5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new RhythmicStaff {
    {
        \time 5/16
        c'8
        c'8
        r16
    }
    {
        r16
        c'16.
        c'16.
        r16
    }
    {
        c'8
        c'8
        r16
    }
    {
        r16
        c'16.
        c'16.
        r16
    }
}
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Methods

`DivisionIncisedNoteRhythmMaker.`**`new`**`(`***kwargs*`)`

Create new division-incised note rhythm-maker with *kwargs*:

```
>>> z(maker)
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
    prefix_talea=[-1],
    prefix_lengths=[0, 1],
    suffix_talea=[-1],
    suffix_lengths=[1],
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
```

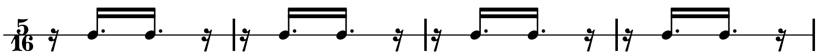
```
        prolation_addenda=[],
        secondary_divisions=[],
        decrease_durations_monotonically=True,
        tie_rests=False,
        beam_each_cell=False,
        beam_cells_together=False
        )
```

```
>>> new_maker = maker.new(prefix_lengths=[1])
```

```
>>> z(new_maker)
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
    prefix_talea=[-1],
    prefix_lengths=[1],
    suffix_talea=[-1],
    suffix_lengths=[1],
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new division-incised note rhythm-maker.

`DivisionIncisedNoteRhythmMaker.`**`reverse`**`()`

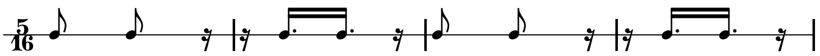
Reverse division-incised note rhythm-maker.

Nonreversed output:

```
>>> z(maker)
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
    prefix_talea=[-1],
    prefix_lengths=[0, 1],
    suffix_talea=[-1],
    suffix_lengths=[1],
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



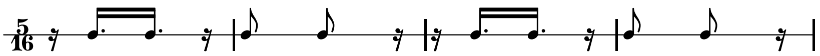
Reversed output:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
    prefix_talea=[-1],
    prefix_lengths=[1, 0],
    suffix_talea=[-1],
    suffix_lengths=[1],
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=False,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return division-incised note rhythm-maker.

DivisionIncisedNoteRhythmMaker.**storage_format**()
Division-incised note rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.DivisionIncisedNoteRhythmMaker(
    prefix_talea=[-1],
    prefix_lengths=[0, 1],
    suffix_talea=[-1],
    suffix_lengths=[1],
    talea_denominator=16,
    body_ratio=mathtools.Ratio(1, 1),
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

Return string.

Special methods

DivisionIncisedNoteRhythmMaker.**__call__**(*divisions*, *seeds=None*)
Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

DivisionIncisedNoteRhythmMaker.**__eq__**(*expr*)
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`DivisionIncisedNoteRhythmMaker.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedNoteRhythmMaker.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`DivisionIncisedNoteRhythmMaker.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedNoteRhythmMaker.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

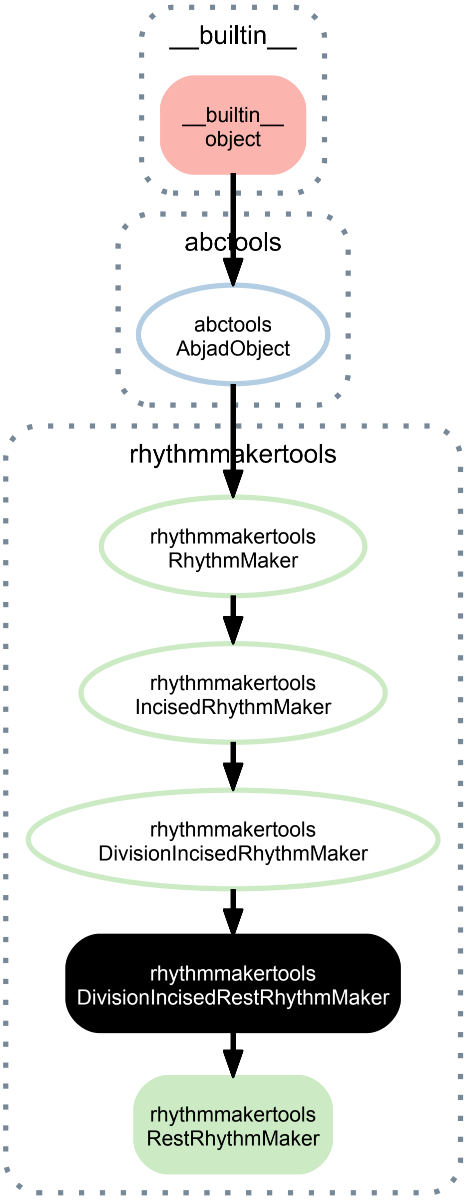
`DivisionIncisedNoteRhythmMaker.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`DivisionIncisedNoteRhythmMaker.__repr__()`
Rhythm-maker interpreter representation.

Return string.

24.2.3 rhythmmakertools.DivisionIncisedRestRhythmMaker



class `rhythmmakertools.`**`DivisionIncisedRestRhythmMaker`**(*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

New in version 2.8. Division-incised rest rhythm-maker:

```
>>> maker = rhythmmakertools.DivisionIncisedRestRhythmMaker(
...     prefix_talea=[1],
...     prefix_lengths=[1, 2, 3, 4],
...     suffix_talea=[1],
...     suffix_lengths=[1],
...     talea_denominator=32)
```

Configure at instantiation and then call on any sequence of divisions:

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```

Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

`DivisionIncisedRestRhythmMaker.`**`storage_format`**

Division-incised rest rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[1, 2, 3, 4],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
```

```
        beam_cells_together=False
        )
```

Return string.

Methods

`DivisionIncisedRestRhythmMaker.`**`new`**`(**kwargs)`
Create new division-incised rest rhythm-maker with *kwargs*:

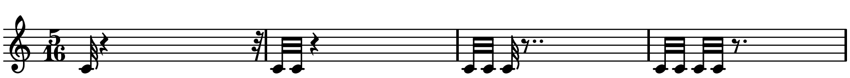
```
>>> z(maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[1, 2, 3, 4],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> new_maker = maker.new(suffix_lengths=[0])
```

```
>>> z(new_maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[1, 2, 3, 4],
    suffix_talea=[1],
    suffix_lengths=[0],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new division-incised rest rhythm-maker.

`DivisionIncisedRestRhythmMaker.`**`reverse`**`()`
Reverse division-incised rest rhythm-maker:

```
>>> z(maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[1, 2, 3, 4],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
```

```
        decrease_durations_monotonically=True,
        tie_rests=False,
        beam_each_cell=False,
        beam_cells_together=False
        )
```

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.DivisionIncisedRestRhythmMaker(
    prefix_talea=[1],
    prefix_lengths=[4, 3, 2, 1],
    suffix_talea=[1],
    suffix_lengths=[1],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=False,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 16), (5, 16), (5, 16), (5, 16)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```

Return new division-incised rest rhythm-maker.

Special methods

`DivisionIncisedRestRhythmMaker.`**`__call__`**`(`*divisions, seeds=None*`)`

Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`DivisionIncisedRestRhythmMaker.`**`__eq__`**`(`*expr*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`DivisionIncisedRestRhythmMaker.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedRestRhythmMaker.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`DivisionIncisedRestRhythmMaker.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`DivisionIncisedRestRhythmMaker.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

DivisionIncisedRestRhythmMaker.**__ne__**(*expr*)

Defined equal to the opposite of equality.

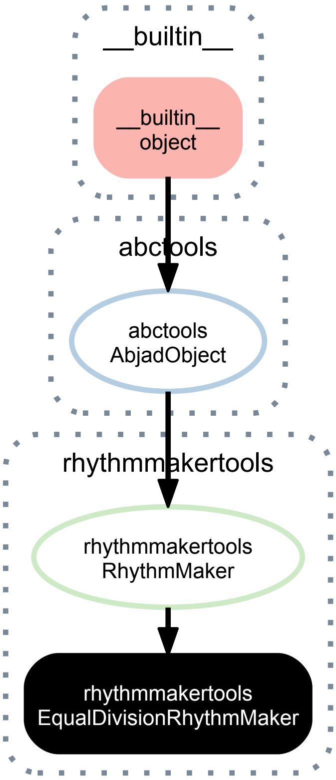
Return boolean.

DivisionIncisedRestRhythmMaker.**__repr__**()

Rhythm-maker interpreter representation.

Return string.

24.2.4 rhythmmakertools.EqualDivisionRhythmMaker



class rhythmmakertools.**EqualDivisionRhythmMaker**(*leaf_count=None*, *is_diminution=True*, *beam_each_cell=True*, *beam_cells_together=False*)

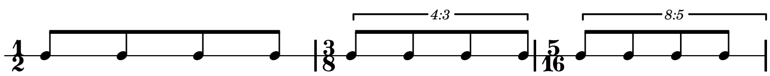
New in version 2.10. Equal division rhythm-maker:

```
>>> maker = rhythmmakertools.EqualDivisionRhythmMaker(leaf_count=4)
```

Configure at initialization and then call on any series of divisions:

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

EqualDivisionRhythmMaker.**is_diminution**
True when output tuplets should be diminuted.

False when output tuplets should be augmented:

```
>>> maker.is_diminution
True
```

Return boolean.

EqualDivisionRhythmMaker.**leaf_count**
Number of leaves per division:

```
>>> maker.leaf_count
4
```

Return positive integer.

EqualDivisionRhythmMaker.**storage_format**
Equal-division rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.EqualDivisionRhythmMaker(
    leaf_count=4,
    is_diminution=True,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

Return string.

Methods

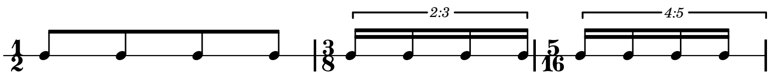
EqualDivisionRhythmMaker.**new**(***kwargs*)
Create new equal-division rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(is_diminution=False)
```

```
>>> z(new_maker)
rhythmmakertools.EqualDivisionRhythmMaker(
    leaf_count=4,
    is_diminution=False,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new equal-division rhythm-maker.

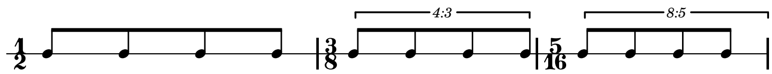
EqualDivisionRhythmMaker.**reverse**()
Reverse equal-division rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.EqualDivisionRhythmMaker(
    leaf_count=4,
    is_diminution=True,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

```
>>> divisions = [(1, 2), (3, 8), (5, 16)]
>>> tuplet_lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(tuplet_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Defined equal to copy of maker.

Return new equal-division rhythm-maker.

Special methods

`EqualDivisionRhythmMaker.__call__(divisions, seeds=None)`
Call equal-division rhythm-maker on *divisions*.

Return list of tuplet lists.

`EqualDivisionRhythmMaker.__eq__(expr)`
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`EqualDivisionRhythmMaker.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`EqualDivisionRhythmMaker.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`EqualDivisionRhythmMaker.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`EqualDivisionRhythmMaker.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

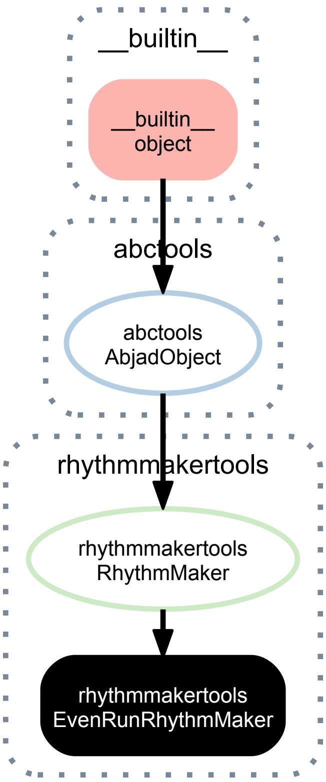
`EqualDivisionRhythmMaker.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`EqualDivisionRhythmMaker.__repr__()`
Rhythm-maker interpreter representation.

Return string.

24.2.5 rhythmmakertools.EvenRunRhythmMaker



class `rhythmmakertools.`**`EvenRunRhythmMaker`**(*denominator_multiplier_exponent=0*, *beam_each_cell=True*, *beam_cells_together=False*)

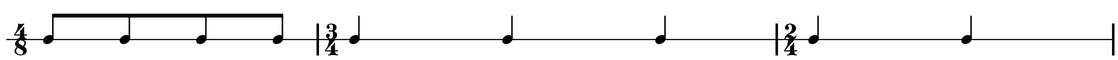
New in version 2.11. Even run rhythm-maker.

Example 1. Make even run of notes each equal in duration to `1/d` with `d` equal to the denominator of each division on which the rhythm-maker is called:

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker()
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```

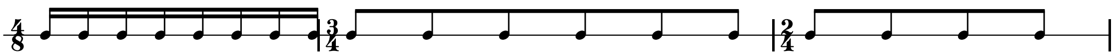


Example 2. Make even run of notes each equal in duration to `1/(2**d)` with `d` equal to the denominator of each division on which the rhythm-maker is called:

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker(1)
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Output a list of lists of depth-2 note-bearing containers.

Even-run rhythm-maker doesn't yet work with non-power-of-two divisions.

Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

`EvenRunRhythmMaker.`**`denominator_multiplier_exponent`**

Denominator multiplier exponent provided at initialization.

```
>>> maker.denominator_multiplier_exponent
1
```

Return nonnegative integer.

`EvenRunRhythmMaker.`**`storage_format`**

Even-run rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=1,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

Return string.

Methods

`EvenRunRhythmMaker.`**`new`**`(**kwargs)`

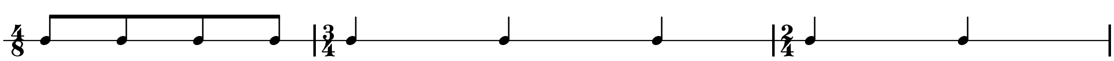
Create new even-run rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(denominator_multiplier_exponent=0)
```

```
>>> z(new_maker)
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=0,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new even-run rhythm-maker.

`EvenRunRhythmMaker.`**`reverse`**`()`

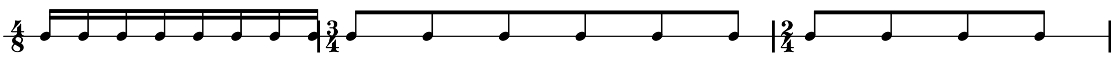
Reverse even-run rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.EvenRunRhythmMaker(
    denominator_multiplier_exponent=1,
    beam_each_cell=True,
    beam_cells_together=False
    )
```

```
>>> divisions = [(4, 8), (3, 4), (2, 4)]
>>> lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Defined equal to copy of even-run rhythm-maker.

Return new even-run rhythm-maker.

Special methods

EvenRunRhythmMaker.**__call__**(*divisions*, *seeds=None*)

Call even-run rhythm-maker on *divisions*.

Return list of container lists.

EvenRunRhythmMaker.**__eq__**(*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

EvenRunRhythmMaker.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

EvenRunRhythmMaker.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

EvenRunRhythmMaker.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

EvenRunRhythmMaker.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

EvenRunRhythmMaker.**__ne__**(*expr*)

Defined equal to the opposite of equality.

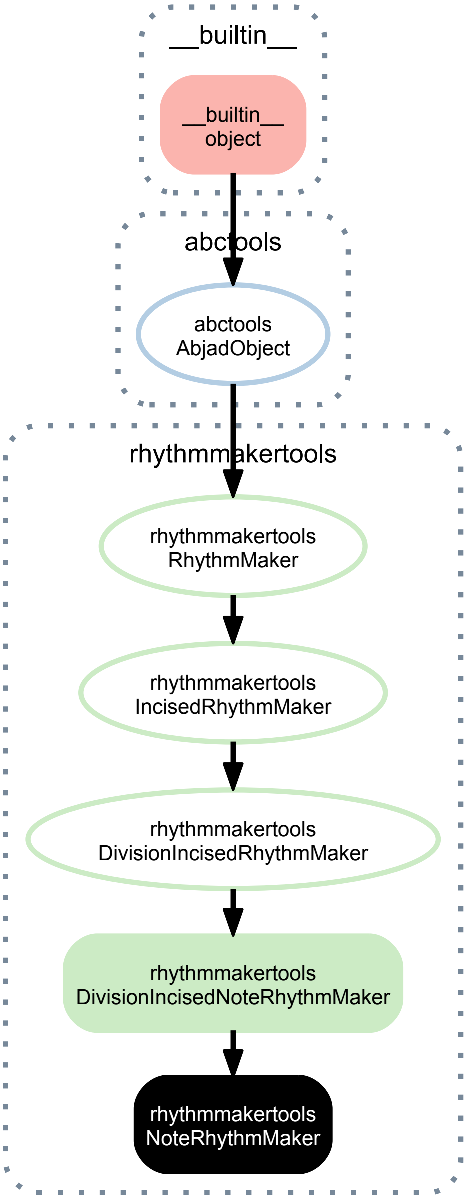
Return boolean.

EvenRunRhythmMaker.**__repr__**()

Rhythm-maker interpreter representation.

Return string.

24.2.6 rhythmmakertools.NoteRhythmMaker



class `rhythmmakertools.`**`NoteRhythmMaker`**(*decrease_durations_monotonically=True, forbidden_written_duration=None, tie_rests=False*)

New in version 2.8. Note rhythm-maker:

Example 1:

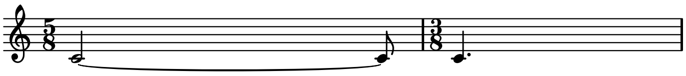
```
>>> maker = rhythmmakertools.NoteRhythmMaker()
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/8
        c'2 ~
        c'8
```

```
        }
        {
            \time 3/8
            c'4.
        }
    }
```

```
>>> show(staff)
```



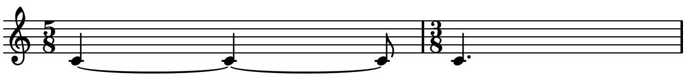
Example 2. Forbid half notes:

```
>>> maker = rhythmmakertools.NoteRhythmMaker(forbidden_written_duration=Duration(1, 2))
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new Staff {
    {
        \time 5/8
        c'4 ~
        c'4 ~
        c'8
    }
    {
        \time 3/8
        c'4.
    }
}
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

NoteRhythmMaker.**storage_format**
Note rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.NoteRhythmMaker(
    decrease_durations_monotonically=True,
    forbidden_written_duration=durationtools.Duration(1, 2),
    tie_rests=False
    )
```

Return string.

Methods

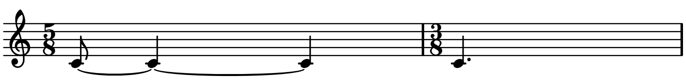
NoteRhythmMaker.**new**(***kwargs*)
Create new note rhythm-maker:

```
>>> new_maker = maker.new(decrease_durations_monotonically=False)
```

```
>>> z(new_maker)
rhythmmakertools.NoteRhythmMaker(
    decrease_durations_monotonically=False,
    forbidden_written_duration=durationtools.Duration(1, 2),
    tie_rests=False
    )
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new note rhythm-maker.

`NoteRhythmMaker.`**`reverse`**`()`

Reverse note rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.NoteRhythmMaker(
    decrease_durations_monotonically=False,
    forbidden_written_duration=durationtools.Duration(1, 2),
    tie_rests=False
    )
```

```
>>> divisions = [(5, 8), (3, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```

Return new note rhythm-maker.

Special methods

`NoteRhythmMaker.`**`__call__`**`(`*divisions*, *seeds=None*`)`

Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`NoteRhythmMaker.`**`__eq__`**`(`*expr*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`NoteRhythmMaker.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`NoteRhythmMaker.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`NoteRhythmMaker.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`NoteRhythmMaker.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

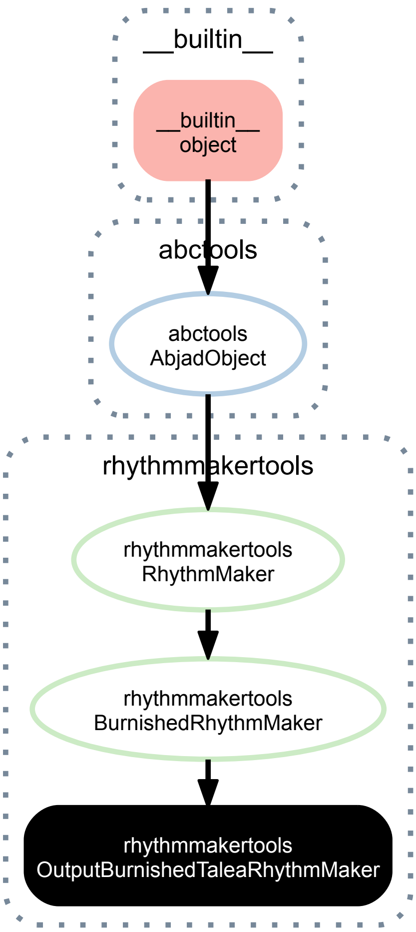
`NoteRhythmMaker.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`NoteRhythmMaker.__repr__()`
Note rhythm-maker interpreter representation.

Return string.

24.2.7 rhythmmakertools.OutputBurnishedTaleaRhythmMaker



class `rhythmmakertools.`**`OutputBurnishedTaleaRhythmMaker`**(*talea=None, talea_denominator=None, prolation_addenda=None, lefts=None, middles=None, rights=None, left_lengths=None, right_lengths=None, secondary_divisions=None, talea_helper=None, prolation_addenda_helper=None, lefts_helper=None, middles_helper=None, rights_helper=None, left_lengths_helper=None, right_lengths_helper=None, secondary_divisions_helper=None, beam_each_cell=False, beam_cells_together=False, decrease_durations_monotonically=True, tie_split_notes=False, tie_rests=False*)

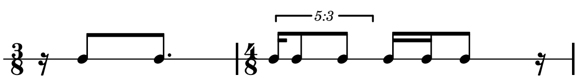
New in version 2.8. Output-burnished talea rhythm-maker:

```
>>> maker = rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
...     talea=[1, 2, 3],
...     talea_denominator=16,
...     prolation_addenda=[0, 2],
...     lefts=[-1],
...     middles=[0],
...     rights=[-1],
...     left_lengths=[1],
...     right_lengths=[1],
...     secondary_divisions=[9],
...     beam_each_cell=True)
```

Configure at initialization and then call on any list of divisions:

```
>>> divisions = [(3, 8), (4, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

`OutputBurnishedTaleaRhythmMaker.`**`storage_format`**

Output-burnished talea rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
    talea=[1, 2, 3],
    talea_denominator=16,
    prolation_addenda=[0, 2],
    lefts=[-1],
```

```
        middles=[0],
        rights=[-1],
        left_lengths=[1],
        right_lengths=[1],
        secondary_divisions=[9],
        beam_each_cell=True,
        beam_cells_together=False,
        decrease_durations_monotonically=True,
        tie_split_notes=False,
        tie_rests=False
        )
```

Return string.

Methods

`OutputBurnishedTaleaRhythmMaker.`**`new`**`(`***kwargs*`)`

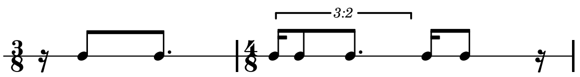
Create new output-burnished talea rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(secondary_divisions=[10])
```

```
>>> z(new_maker)
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
    talea=[1, 2, 3],
    talea_denominator=16,
    prolation_addenda=[0, 2],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
    secondary_divisions=[10],
    beam_each_cell=True,
    beam_cells_together=False,
    decrease_durations_monotonically=True,
    tie_split_notes=False,
    tie_rests=False
    )
```

```
>>> divisions = [(3, 8), (4, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new output-burnished talea rhythm-maker.

`OutputBurnishedTaleaRhythmMaker.`**`reverse`**`()`

Reverse output-burnished talea rhythm-maker:

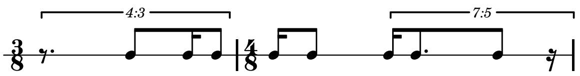
```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.OutputBurnishedTaleaRhythmMaker(
    talea=[3, 2, 1],
    talea_denominator=16,
    prolation_addenda=[2, 0],
    lefts=[-1],
    middles=[0],
    rights=[-1],
    left_lengths=[1],
    right_lengths=[1],
```

```
        secondary_divisions=[9],
        beam_each_cell=True,
        beam_cells_together=False,
        decrease_durations_monotonically=False,
        tie_split_notes=False,
        tie_rests=False
        )
```

```
>>> divisions = [(3, 8), (4, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new output-burnished talea rhythm-maker.

Special methods

`OutputBurnishedTaleaRhythmMaker.`**`__call__`**(*divisions*, *seeds=None*)

Call burnished rhythm-maker on *divisions*.

Return either list of tuplets or else list of note-lists.

`OutputBurnishedTaleaRhythmMaker.`**`__eq__`**(*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`OutputBurnishedTaleaRhythmMaker.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`OutputBurnishedTaleaRhythmMaker.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`OutputBurnishedTaleaRhythmMaker.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`OutputBurnishedTaleaRhythmMaker.`**`__lt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`OutputBurnishedTaleaRhythmMaker.`**`__ne__`**(*expr*)

Defined equal to the opposite of equality.

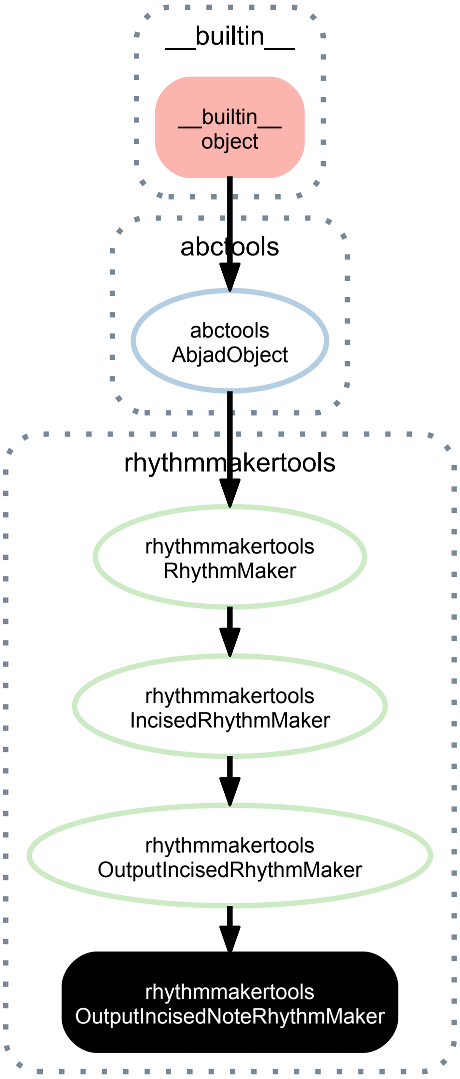
Return boolean.

`OutputBurnishedTaleaRhythmMaker.`**`__repr__`**()

Rhythm-maker interpreter representation.

Return string.

24.2.8 rhythmmakertools.OutputIncisedNoteRhythmMaker



class `rhythmmakertools.`**`OutputIncisedNoteRhythmMaker`**(*prefix_talea=None, prefix_lengths=None, suffix_talea=None, suffix_lengths=None, talea_denominator=None, body_ratio=None, prolation_addenda=None, secondary_divisions=None, prefix_talea_helper=None, prefix_lengths_helper=None, suffix_talea_helper=None, suffix_lengths_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, decrease_durations_monotonically=True, tie_rests=False, forbidden_written_duration=None, beam_each_cell=False, beam_cells_together=False*)

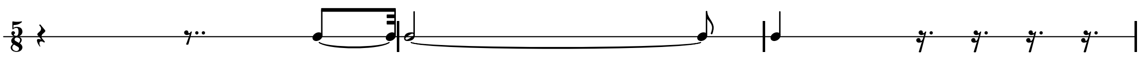
New in version 2.8. Output-incised note rhythm-maker:

```
>>> maker = rhythmmakertools.OutputIncisedNoteRhythmMaker(
...     prefix_talea=[-8, -7],
...     prefix_lengths=[2],
...     suffix_talea=[-3],
...     suffix_lengths=[4],
...     talea_denominator=32)
```

Configure at initialization and then call on arbitrary divisions:

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

OutputIncisedNoteRhythmMaker.**storage_format**

Output-incised note rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.OutputIncisedNoteRhythmMaker(
    prefix_talea=[-8, -7],
    prefix_lengths=[2],
    suffix_talea=[-3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

Return string.

Methods

OutputIncisedNoteRhythmMaker.**new**(***kwargs*)

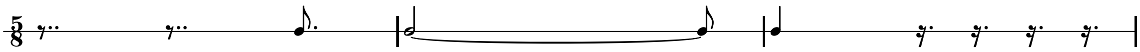
Create new output-incised note rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(prefix_talea=[-7])
```

```
>>> z(new_maker)
rhythmmakertools.OutputIncisedNoteRhythmMaker(
    prefix_talea=[-7],
    prefix_lengths=[2],
    suffix_talea=[-3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new output-incised note rhythm-maker.

`OutputIncisedNoteRhythmMaker.`**`reverse`**`()`

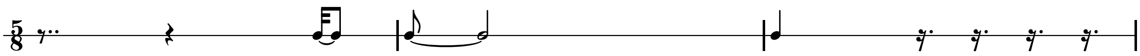
Reverse output-incised note rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.OutputIncisedNoteRhythmMaker(
    prefix_talea=[-7, -8],
    prefix_lengths=[2],
    suffix_talea=[-3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=False,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new output-incised note rhythm-maker.

Special methods

`OutputIncisedNoteRhythmMaker.`**`__call__`**`(`*divisions*, *seeds=None*`)`

Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

`OutputIncisedNoteRhythmMaker.`**`__eq__`**`(`*expr*`)`

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`OutputIncisedNoteRhythmMaker.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`OutputIncisedNoteRhythmMaker.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`OutputIncisedNoteRhythmMaker.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`OutputIncisedNoteRhythmMaker.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

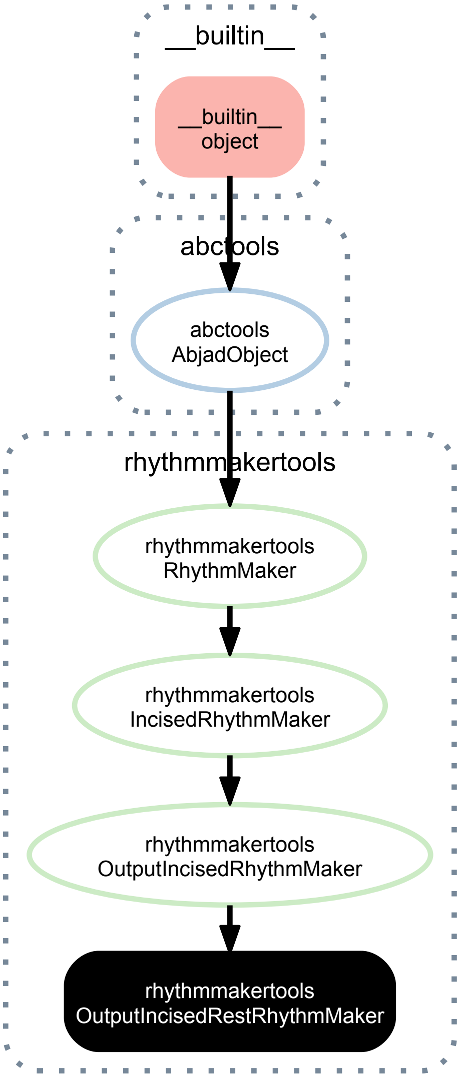
`OutputIncisedNoteRhythmMaker.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`OutputIncisedNoteRhythmMaker.__repr__()`
Rhythm-maker interpreter representation.

Return string.

24.2.9 rhythmmakertools.OutputIncisedRestRhythmMaker



class `rhythmmakertools.`**`OutputIncisedRestRhythmMaker`**(*prefix_talea=None*, *prefix_lengths=None*, *suffix_talea=None*, *suffix_lengths=None*, *talea_denominator=None*, *body_ratio=None*, *prolation_addenda=None*, *secondary_divisions=None*, *prefix_talea_helper=None*, *prefix_lengths_helper=None*, *suffix_talea_helper=None*, *suffix_lengths_helper=None*, *prolation_addenda_helper=None*, *secondary_divisions_helper=None*, *decrease_durations_monotonically=True*, *tie_rests=False*, *forbidden_written_duration=None*, *beam_each_cell=False*, *beam_cells_together=False*)

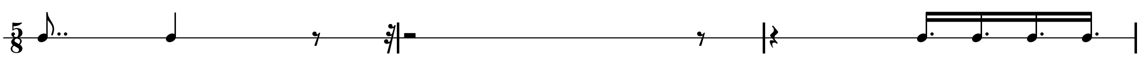
New in version 2.8. Output-incised rest rhythm-maker:

```
>>> maker = rhythmmakertools.OutputIncisedRestRhythmMaker(
...     prefix_talea=[7, 8],
...     prefix_lengths=[2],
...     suffix_talea=[3],
...     suffix_lengths=[4],
...     talea_denominator=32)
```

Configuration at initialization and then call on arbitrary divisions:

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

`OutputIncisedRestRhythmMaker.`**`storage_format`**

Output-incised rest rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.OutputIncisedRestRhythmMaker(
    prefix_talea=[7, 8],
    prefix_lengths=[2],
    suffix_talea=[3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

Return string.

Methods

`OutputIncisedRestRhythmMaker.new(**kwargs)`

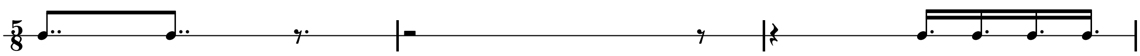
Create new output-incised rest rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new(prefix_talea=[7])
```

```
>>> z(new_maker)
rhythmmakertools.OutputIncisedRestRhythmMaker(
    prefix_talea=[7],
    prefix_lengths=[2],
    suffix_talea=[3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=True,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new output-incised rest rhythm-maker.

`OutputIncisedRestRhythmMaker.reverse()`

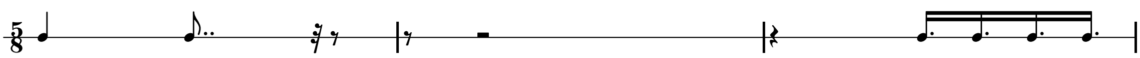
Reverse output-incised rest rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.OutputIncisedRestRhythmMaker(
    prefix_talea=[8, 7],
    prefix_lengths=[2],
    suffix_talea=[3],
    suffix_lengths=[4],
    talea_denominator=32,
    prolation_addenda=[],
    secondary_divisions=[],
    decrease_durations_monotonically=False,
    tie_rests=False,
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(5, 8), (5, 8), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new output-incised rest rhythm-maker.

Special methods

OutputIncisedRestRhythmMaker.**__call__**(*divisions*, *seeds=None*)
Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

OutputIncisedRestRhythmMaker.**__eq__**(*expr*)
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

OutputIncisedRestRhythmMaker.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OutputIncisedRestRhythmMaker.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

OutputIncisedRestRhythmMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OutputIncisedRestRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

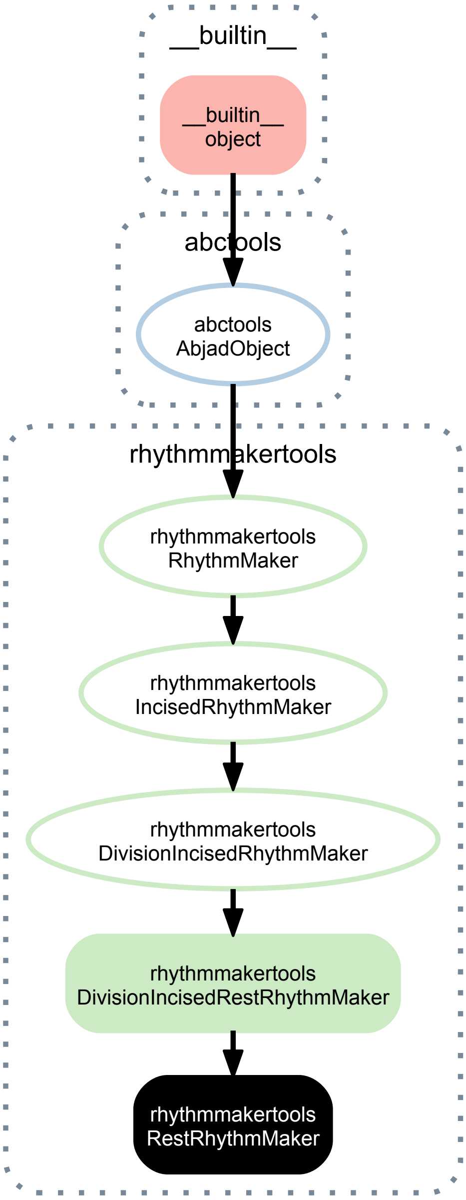
OutputIncisedRestRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

OutputIncisedRestRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

24.2.10 rhythmmakertools.RestRhythmMaker



class `rhythmmakertools.`**`RestRhythmMaker`**(*forbidden_written_duration=None*)

New in version 2.8. Rest rhythm-maker.

Example 1:

```
>>> maker = rhythmmakertools.RestRhythmMaker()
```

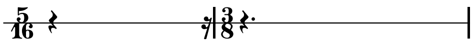
Initialize and then call on arbitrary divisions:

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new RhythmicStaff {
    {
        \time 5/16
        r4
```

```
            r16
        }
        {
            \time 3/8
            r4.
        }
    }
```

```
>>> show(staff)
```



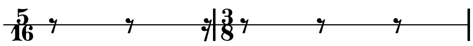
Example 2. Forbid written durations greater than or equal to a half note:

```
>>> maker = rhythmmakertools.RestRhythmMaker(forbidden_written_duration=Duration(1, 4))
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> f(staff)
\new RhythmicStaff {
    {
        \time 5/16
        r8
        r8
        r16
    }
    {
        \time 3/8
        r8
        r8
        r8
    }
}
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

RestRhythmMaker.**storage_format**
Rest rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
    )
```

Return string.

Methods

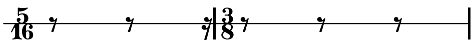
RestRhythmMaker.**new**(***kwargs*)
Create new rest rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> z(new_maker)
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
    )
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = new_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new rest rhythm-maker.

RestRhythmMaker.**reverse**()

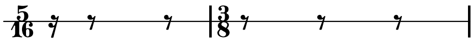
Reverse rest rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.RestRhythmMaker(
    forbidden_written_duration=durationtools.Duration(1, 4)
    )
```

```
>>> divisions = [(5, 16), (3, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, leaves)
```

```
>>> show(staff)
```



Return new rest rhythm-maker.

Special methods

RestRhythmMaker.**__call__**(*divisions*, *seeds=None*)

Call incised rhythm-maker on *divisions*.

Return list of tuplets or return list of leaf lists.

RestRhythmMaker.**__eq__**(*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

RestRhythmMaker.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

RestRhythmMaker.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

RestRhythmMaker.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`RestRhythmMaker.__lt__(expr)`

Abjad objects by default do not implement this method.

Raise exception.

`RestRhythmMaker.__ne__(expr)`

Defined equal to the opposite of equality.

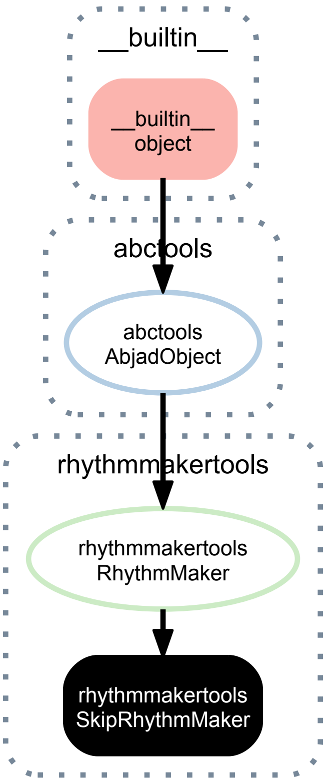
Return boolean.

`RestRhythmMaker.__repr__()`

Rhythm-maker interpreter representation.

Return string.

24.2.11 rhythmmakertools.SkipRhythmMaker



class `rhythmmakertools.SkipRhythmMaker`

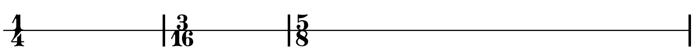
New in version 2.10. Skip rhythm-maker:

```
>>> maker = rhythmmakertools.SkipRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

SkipRhythmMaker.**storage_format**

Skip rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.SkipRhythmMaker()
```

Return string.

Methods

SkipRhythmMaker.**new**(***kwargs*)

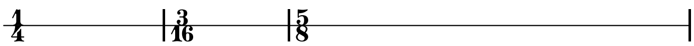
Create new skip rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> z(new_maker)
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new skip rhythm-maker.

SkipRhythmMaker.**reverse**()

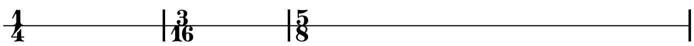
Reverse skip rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.SkipRhythmMaker()
```

```
>>> divisions = [(1, 4), (3, 16), (5, 8)]
>>> leaf_lists = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(leaf_lists)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = stafftools.RhythmicStaff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new skip rhythm-maker.

Special methods

SkipRhythmMaker.**__call__**(*divisions*, *seeds=None*)

Call skip rhythm-maker on *divisions*.

Return list of skips.

SkipRhythmMaker.**__eq__**(*expr*)

True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

SkipRhythmMaker.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SkipRhythmMaker.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SkipRhythmMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SkipRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

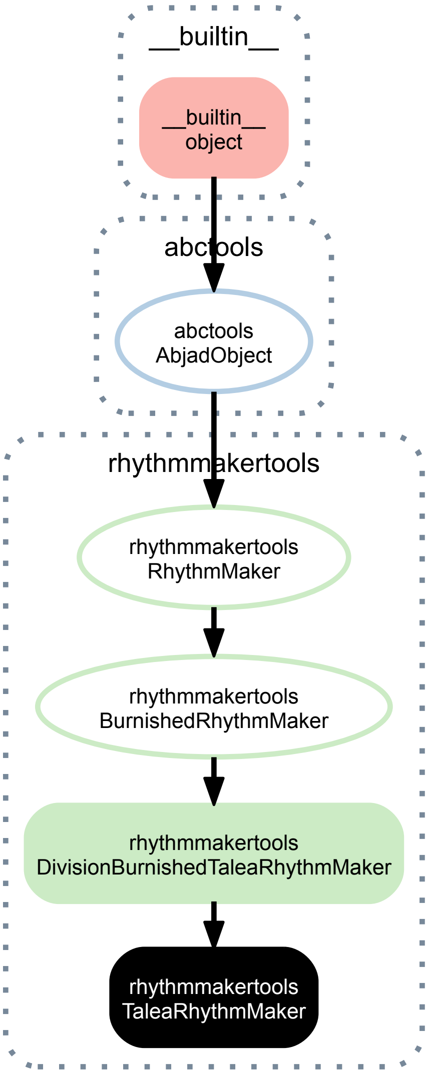
SkipRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SkipRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

24.2.12 rhythmmakertools.TaleaRhythmMaker



class `rhythmmakertools.`**`TaleaRhythmMaker`**(*talea=None, talea_denominator=None, prolation_addenda=None, secondary_divisions=None, talea_helper=None, prolation_addenda_helper=None, secondary_divisions_helper=None, beam_each_cell=False, beam_cells_together=False, tie_split_notes=False*)

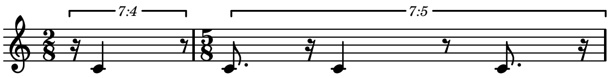
New in version 2.8. Talea rhythm-maker.

Example 1. Basic usage:

```
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=[-1, 4, -2, 3],
...     talea_denominator=16,
...     prolation_addenda=[3, 4])
```

```
>>> divisions = [(2, 8), (5, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```

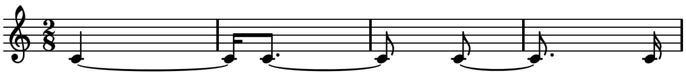


Example 2. Tie split notes.

```
>>> maker = rhythmmakertools.TaleaRhythmMaker(
...     talea=[5],
...     talea_denominator=16,
...     tie_split_notes=True)
```

```
>>> divisions = [(2, 8), (2, 8), (2, 8), (2, 8)]
>>> music = maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

TaleaRhythmMaker.**storage_format**
Talea rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.TaleaRhythmMaker(
    talea=[5],
    talea_denominator=16,
    prolation_addenda=[],
    secondary_divisions=[],
    beam_each_cell=False,
    beam_cells_together=False,
    tie_split_notes=True
    )
```

Return string.

Methods

TaleaRhythmMaker.**new**(***kwargs*)
Create new talea rhythm-maker with *kwargs*:

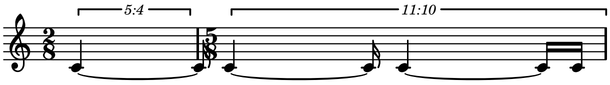
```
>>> new_maker = maker.new(prolation_addenda=[1])
```

```
>>> z(new_maker)
rhythmmakertools.TaleaRhythmMaker(
    talea=[5],
    talea_denominator=16,
    prolation_addenda=[1],
    secondary_divisions=[],
    beam_each_cell=False,
    beam_cells_together=False,
    tie_split_notes=True
    )
```

```
>>> divisions = [(2, 8), (5, 8)]
>>> music = new_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
```

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```



Return new talea rhythm-maker.

`TaleaRhythmMaker.reverse()`
Reverse talea rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.TaleaRhythmMaker(
    talea=[5],
    talea_denominator=16,
    prolation_addenda=[],
    secondary_divisions=[],
    beam_each_cell=False,
    beam_cells_together=False,
    tie_split_notes=True
    )
```

```
>>> divisions = [(2, 8), (5, 8)]
>>> music = reversed_maker(divisions)
>>> music = sequencetools.flatten_sequence(music)
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(divisions)
>>> staff = Staff(measures)
>>> measures = measuretools.replace_contents_of_measures_in_expr(staff, music)
```

```
>>> show(staff)
```

Return new talea rhythm-maker.

Special methods

`TaleaRhythmMaker.__call__(`*divisions, seeds=None*`)`
Call burnished rhythm-maker on *divisions*.

Return either list of tuplets or else list of note-lists.

`TaleaRhythmMaker.__eq__(`*expr*`)`
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

`TaleaRhythmMaker.__ge__(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TaleaRhythmMaker.__gt__(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`TaleaRhythmMaker.__le__(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

TaleaRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

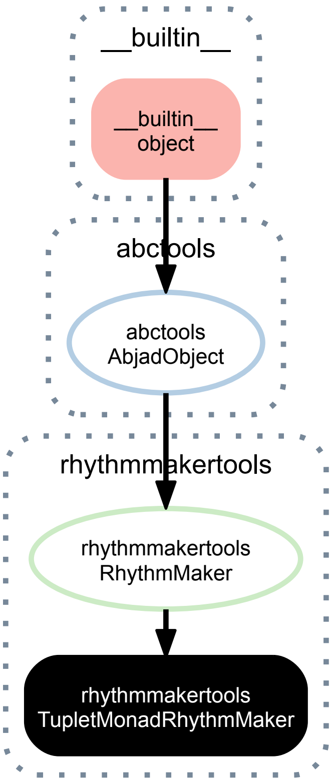
TaleaRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TaleaRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

24.2.13 rhythmmakertools.TupletMonadRhythmMaker



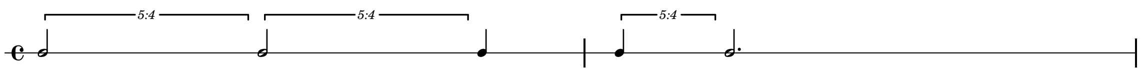
class rhythmmakertools.**TupletMonadRhythmMaker**(*beam_each_cell=False*, *beam_cells_together=False*)
New in version 2.10. Tuplet monad rhythm-maker:

```
>>> maker = rhythmmakertools.TupletMonadRhythmMaker()
```

Initialize and then call on arbitrary divisions:

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Usage follows the two-step instantiate-then-call pattern shown here.

Read-only properties

TupletMonadRhythmMaker.**storage_format**

Tuplet monad rhythm-maker storage format:

```
>>> z(maker)
rhythmmakertools.TupletMonadRhythmMaker(
    beam_each_cell=False,
    beam_cells_together=False
    )
```

Return string.

Methods

TupletMonadRhythmMaker.**new**(***kwargs*)

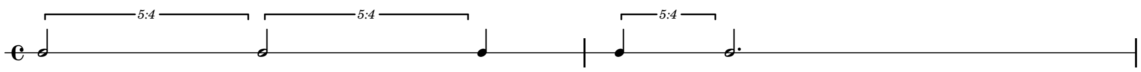
Create new tuplet monad rhythm-maker with *kwargs*:

```
>>> new_maker = maker.new()
```

```
>>> z(new_maker)
rhythmmakertools.TupletMonadRhythmMaker(
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = new_maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Return new tuplet monad rhythm-maker.

TupletMonadRhythmMaker.**reverse**()

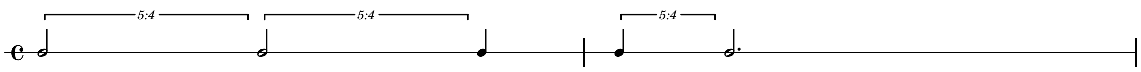
Reverse tuplet monad rhythm-maker:

```
>>> reversed_maker = maker.reverse()
```

```
>>> z(reversed_maker)
rhythmmakertools.TupletMonadRhythmMaker(
    beam_each_cell=False,
    beam_cells_together=False
    )
```

```
>>> divisions = [(2, 5), (2, 5), (1, 4), (1, 5), (3, 4)]
>>> tuplet_lists = reversed_maker(divisions)
>>> tuplets = sequencetools.flatten_sequence(tuplet_lists)
>>> staff = stafftools.RhythmicStaff(tuplets)
```

```
>>> show(staff)
```



Return new tuplet monad rhythm-maker.

Special methods

TupletMonadRhythmMaker.**__call__**(*divisions*, *seeds=None*)

Call tuplet monad rhythm-maker on *divisions*.

Return list of tuplets.

TupletMonadRhythmMaker.**__eq__**(*expr*)
True when *expr* is same type with the equal public nonhelper properties. Otherwise false.

Return boolean.

TupletMonadRhythmMaker.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TupletMonadRhythmMaker.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TupletMonadRhythmMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TupletMonadRhythmMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TupletMonadRhythmMaker.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TupletMonadRhythmMaker.**__repr__**()
Rhythm-maker interpreter representation.

Return string.

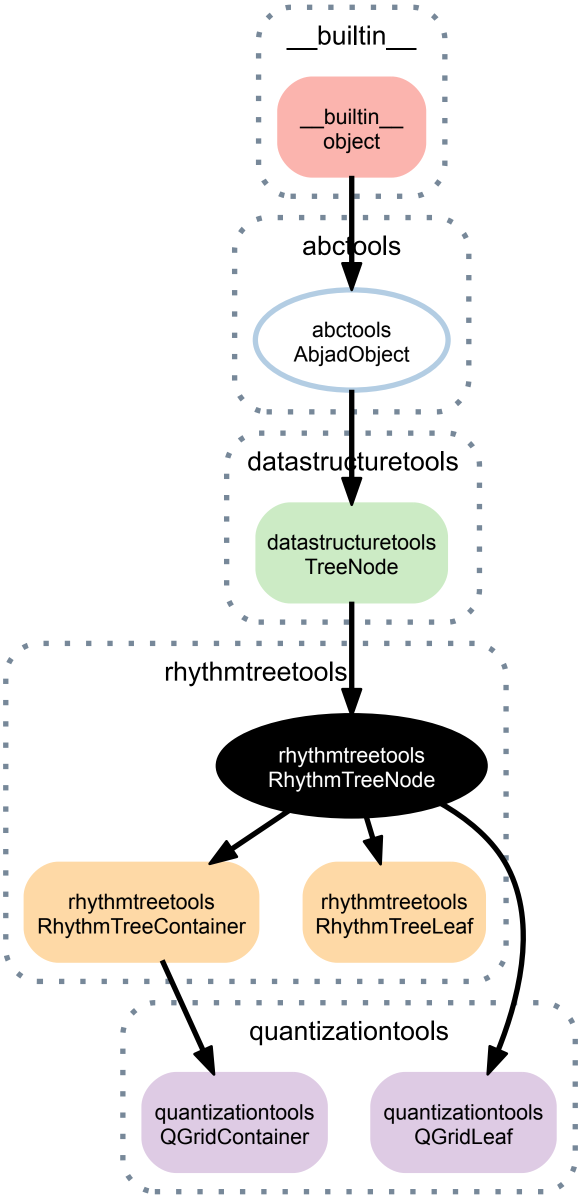
CHAPTER

TWENTYFIVE

RHYTHMTREETOOLS

25.1 Abstract Classes

25.1.1 rhythmtreetools.RhythmTreeNode



class `rhythmtreetools.RhythmTreeNode` (*preprolated_duration=1*, *name=None*)
Abstract base class of nodes in a rhythm tree structure.

Read-only properties

`RhythmTreeNode.depth`
The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`RhythmTreeNode.depthwise_inventory`
A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

`RhythmTreeNode.duration`
The prolated preprolated_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

RhythmTreeNode.**graph_order**

RhythmTreeNode.**graphviz_format**

RhythmTreeNode.**graphviz_graph**

RhythmTreeNode.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeNode.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

RhythmTreeNode.**parentage_ratios**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated_duration of the root node, and subsequent items are pairs of the preprolated_duration of the next node in the parentage chain and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Return tuple.

RhythmTreeNode.**pretty_rtm_format**

The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
    (1 (
        1
        1))
    (1 (
        1
        1))))
```

Return string.

RhythmTreeNode.**prolation**

RhythmTreeNode.**prolations**

RhythmTreeNode.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeNode.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

RhythmTreeNode.**rtm_format**
The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Return string.

RhythmTreeNode.**start_offset**
The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Return Offset instance.

RhythmTreeNode.**stop_offset**
The stopping offset of a node in a rhythm-tree relative the root.

RhythmTreeNode.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

RhythmTreeNode.**name**

RhythmTreeNode.**preprolated_duration**
The node's preprolated_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Return int.

Special methods

RhythmTreeNode.**__call__**(*pulse_duration*)

RhythmTreeNode.**__copy__**(**args*)

RhythmTreeNode.**__deepcopy__**(**args*)

RhythmTreeNode.**__eq__**(*expr*)

RhythmTreeNode.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmTreeNode.**__getstate__**()

RhythmTreeNode.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

RhythmTreeNode.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmTreeNode.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

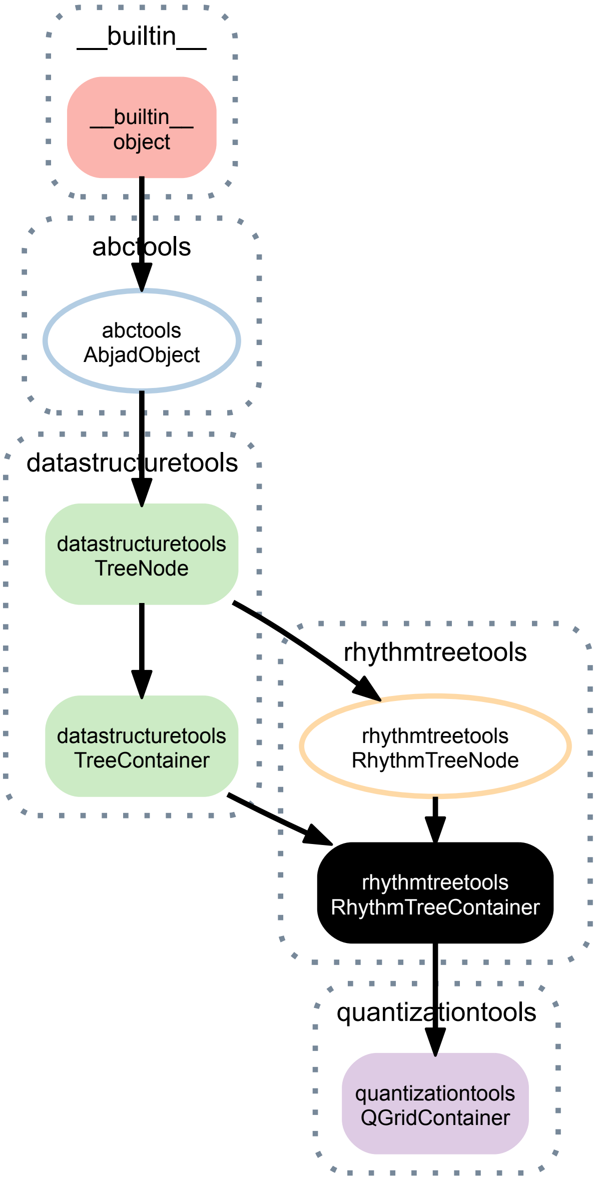
RhythmTreeNode.**__ne__**(*expr*)

RhythmTreeNode.**__repr__**()

RhythmTreeNode.**__setstate__**(*state*)

25.2 Concrete Classes

25.2.1 rhythmtreetools.RhythmTreeContainer



class `rhythmtreetools.RhythmTreeContainer`(*children=None*, *preprolated_duration=1*, *name=None*)

A container node in a rhythm tree structure:

```
>>> container = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1, children=[])
>>> container
RhythmTreeContainer(
    preprolated_duration=Duration(1, 1)
    )
```

Similar to Abjad containers, *RhythmTreeContainer* supports a list interface, and can be appended, extended, indexed and so forth by other *RhythmTreeNode* subclasses:

```
>>> leaf_a = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> leaf_b = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=2)
>>> container.extend([leaf_a, leaf_b])
>>> container
RhythmTreeContainer(
    children=(
```

```
            RhythmTreeLeaf(
                preprolated_duration=Duration(1, 1),
                is_pitched=True
                ),
            RhythmTreeLeaf(
                preprolated_duration=Duration(2, 1),
                is_pitched=True
                )
            ),
        preprolated_duration=Duration(1, 1)
        )
```

```
>>> another_container = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> another_container.append(rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3))
>>> another_container.append(container[1])
>>> container.append(another_container)
>>> container
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeContainer(
            children=(
                RhythmTreeLeaf(
                    preprolated_duration=Duration(3, 1),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(2, 1),
                    is_pitched=True
                    )
                ),
            preprolated_duration=Duration(2, 1)
            )
        ),
    preprolated_duration=Duration(1, 1)
    )
```

Call *RhythmTreeContainer* with a preprolated_duration to generate a tuplet structure:

```
>>> container((1, 4))
[FixedDurationTuplet(1/4, [c'8, {@ 5:4 c'8., c'8 @}])]
```

```
>>> f(_[0])
\times 2/3 {
    c'8
    \times 4/5 {
        c'8.
        c'8
    }
}
```

Returns *RhythmTreeContainer* instance.

Read-only properties

RhythmTreeContainer.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`RhythmTreeContainer.`**`contents_duration`**

The total preprolated_duration of the children of a *RhythmTreeContainer* instance:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.contents_duration
Duration(5, 1)
```

```
>>> tree[1].contents_duration
Duration(3, 1)
```

Return int.

`RhythmTreeContainer.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`RhythmTreeContainer.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
```

```
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

`RhythmTreeContainer.`**`duration`**

The prolated preprolated_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

`RhythmTreeContainer.`**`graph_order`**

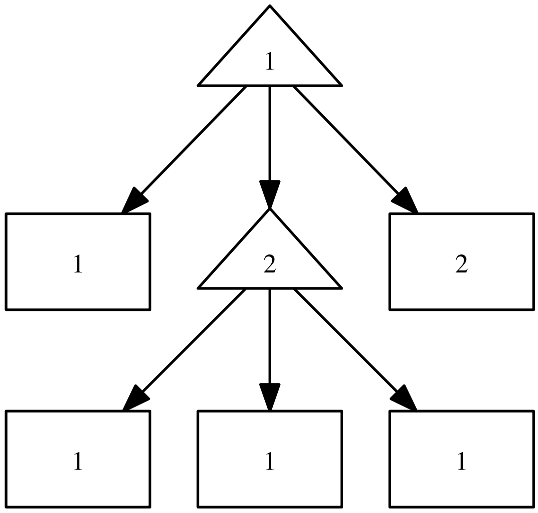
`RhythmTreeContainer.`**`graphviz_format`**

`RhythmTreeContainer.`**`graphviz_graph`**

The GraphvizGraph representation of the RhythmTreeContainer:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> graph = tree.graphviz_graph
>>> print graph.graphviz_format
digraph G {
    node_0 [label=1,
        shape=triangle];
    node_1 [label=1,
        shape=box];
    node_2 [label=2,
        shape=triangle];
    node_3 [label=1,
        shape=box];
    node_4 [label=1,
        shape=box];
    node_5 [label=1,
        shape=box];
    node_6 [label=2,
        shape=box];
    node_0 -> node_1;
    node_0 -> node_2;
    node_0 -> node_6;
    node_2 -> node_3;
    node_2 -> node_4;
    node_2 -> node_5;
}
```

```
>>> iotools.graph(graph)
```



Return *GraphvizGraph* instance.

RhythmTreeContainer.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeContainer.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

RhythmTreeContainer.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`RhythmTreeContainer.`**`parent`**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`RhythmTreeContainer.`**`parentage_ratios`**

A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated_duration of the root node, and subsequent items are pairs of the preprolated_duration of the next node in the parentage chain and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Return tuple.

RhythmTreeContainer.**pretty_rtm_format**
The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
    (1 (
        1
        1))
    (1 (
        1
        1))))
```

Return string.

RhythmTreeContainer.**prolation**

RhythmTreeContainer.**prolations**

RhythmTreeContainer.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeContainer.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

RhythmTreeContainer.**rtm_format**
The node's RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> tree.rtm_format
'(1 ((1 (1 1)) (1 (1 1))))'
```

Return string.

RhythmTreeContainer.**start_offset**
The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Return Offset instance.

RhythmTreeContainer.**stop_offset**
The stopping offset of a node in a rhythm-tree relative the root.

RhythmTreeContainer.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

RhythmTreeContainer.**name**

RhythmTreeContainer.**preprolated_duration**
The node's preprolated_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Return int.

Methods

RhythmTreeContainer.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`RhythmTreeContainer.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`RhythmTreeContainer.`**`index`**`(`*node*`)`

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`RhythmTreeContainer.`**`insert`**`(`*i*, *node*`)`

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
```

```
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`RhythmTreeContainer.`**`pop`**`(`*i=-1*`)`

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

`RhythmTreeContainer.`**`remove`**`(`*node*`)`

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
```

```
            TreeNode()
            )
        )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

RhythmTreeContainer.**__add__**(*expr*)

Concatenate containers self and expr. The operation c = a + b returns a new RhythmTreeContainer c with the content of both a and b, and a preprolated_duration equal to the sum of the durations of a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand:

```
>>> a = rhythmtreetools.RhythmTreeParser()('(1 (1 1 1))')[0]
>>> b = rhythmtreetools.RhythmTreeParser()('(2 (3 4))')[0]
```

```
>>> c = a + b
```

```
>>> c.preprolated_duration
Duration(3, 1)
```

```
>>> c
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(3, 1),
            is_pitched=True
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(4, 1),
            is_pitched=True
            )
        ),
    preprolated_duration=Duration(3, 1)
    )
```

Return new RhythmTreeContainer.

RhythmTreeContainer.**__call__**(*pulse_duration*)

Generate Abjad score components:

```
>>> rtm = '(1 (1 (2 (1 1 1)) 2))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree((1, 4))
[FixedDurationTuplet(1/4, [c'16, {@ 3:2 c'16, c'16, c'16 @}, c'8])]
```

Return sequence of components.

RhythmTreeContainer.**__contains__**(*expr*)
True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

RhythmTreeContainer.**__copy__**(**args*)

RhythmTreeContainer.**__deepcopy__**(**args*)

RhythmTreeContainer.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

RhythmTreeContainer.**__eq__**(*expr*)
True if type, preprolated_duration and children are equivalent, otherwise False.

Return boolean.

RhythmTreeContainer.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmTreeContainer.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

RhythmTreeContainer.**__getstate__**()

RhythmTreeContainer.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

RhythmTreeContainer.**__iter__**()

RhythmTreeContainer.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmTreeContainer.**__len__**()
Return nonnegative integer number of nodes in container.

RhythmTreeContainer.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmTreeContainer.**__ne__**(*expr*)

RhythmTreeContainer.**__repr__**()

RhythmTreeContainer.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = rhythmtreetools.RhythmTreeContainer()
>>> b = rhythmtreetools.RhythmTreeLeaf()
>>> c = rhythmtreetools.RhythmTreeLeaf()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

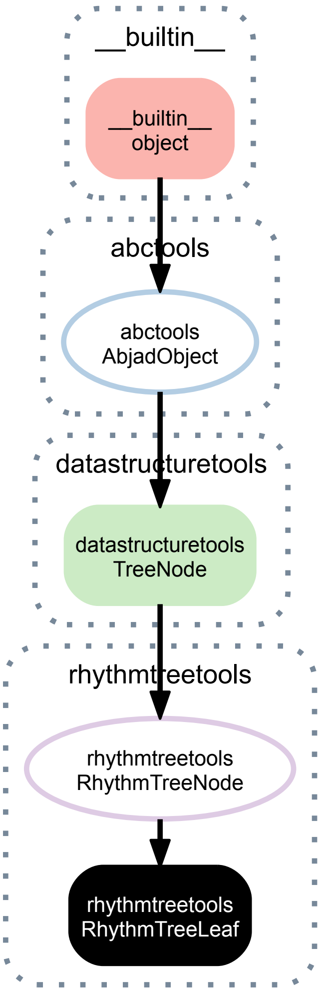
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

`RhythmTreeContainer.`**`__setstate__`**(*state*)

25.2.2 rhythmtreetools.RhythmTreeLeaf



class `rhythmtreetools.`**`RhythmTreeLeaf`**(*preprolated_duration=1*, *is_pitched=True*, *name=None*)

A leaf node in a rhythm tree:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5, is_pitched=True)
>>> leaf
RhythmTreeLeaf(
    preprolated_duration=Duration(5, 1),
    is_pitched=True
    )
```

Call with a pulse preprolated_duration to generate Abjad leaf objects:

```
>>> result = leaf((1, 8))
>>> result
[Note("c'2"), Note("c'8")]
```

Generates rests when called, if *is_pitched* is False:

```
>>> rhythmtreetools.RhythmTreeLeaf(preprolated_duration=7, is_pitched=False)((1, 16))
[Rest('r4..')]
```

Return *RhythmTreeLeaf*.

Read-only properties

RhythmTreeLeaf.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

RhythmTreeLeaf.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
```

```
DEPTH: 2
d
e
f
g
```

Return dictionary.

RhythmTreeLeaf.**duration**

The prolated preprolated_duration of the node:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.duration
Duration(1, 1)
```

```
>>> tree[1].duration
Duration(1, 2)
```

```
>>> tree[1][1].duration
Duration(1, 4)
```

Return *Duration* instance.

RhythmTreeLeaf.**graph_order**

RhythmTreeLeaf.**graphviz_format**

RhythmTreeLeaf.**graphviz_graph**

RhythmTreeLeaf.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeLeaf.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

RhythmTreeLeaf.**parentage_ratios**
A sequence describing the relative durations of the nodes in a node's improper parentage.

The first item in the sequence is the preprolated_duration of the root node, and subsequent items are pairs of the preprolated_duration of the next node in the parentage chain and the total preprolated_duration of that node and its siblings:

```
>>> a = rhythmtreetools.RhythmTreeContainer(preprolated_duration=1)
>>> b = rhythmtreetools.RhythmTreeContainer(preprolated_duration=2)
>>> c = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=3)
>>> d = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=4)
>>> e = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=5)
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.parentage_ratios
(Duration(1, 1),)
```

```
>>> b.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)))
```

```
>>> c.parentage_ratios
(Duration(1, 1), (Duration(3, 1), Duration(5, 1)))
```

```
>>> d.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(4, 1), Duration(9, 1)))
```

```
>>> e.parentage_ratios
(Duration(1, 1), (Duration(2, 1), Duration(5, 1)), (Duration(5, 1), Duration(9, 1)))
```

Return tuple.

RhythmTreeLeaf.**pretty_rtm_format**
The node's pretty-printed RTM format:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
>>> print tree.pretty_rtm_format
(1 (
    (1 (
        1
        1))
    (1 (
        1
        1))))
```

Return string.

RhythmTreeLeaf.**prolation**

RhythmTreeLeaf.**prolations**

RhythmTreeLeaf.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

RhythmTreeLeaf.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

RhythmTreeLeaf.**rtm_format**
The node's RTM format:

```
>>> rhythmtreetools.RhythmTreeLeaf(1, is_pitched=True).rtm_format
'1'
>>> rhythmtreetools.RhythmTreeLeaf(5, is_pitched=False).rtm_format
'-5'
```

Return string.

RhythmTreeLeaf.**start_offset**
The starting offset of a node in a rhythm-tree relative the root:

```
>>> rtm = '(1 ((1 (1 1)) (1 (1 1))))'
>>> tree = rhythmtreetools.RhythmTreeParser()(rtm)[0]
```

```
>>> tree.start_offset
Offset(0, 1)
```

```
>>> tree[1].start_offset
Offset(1, 2)
```

```
>>> tree[0][1].start_offset
Offset(1, 4)
```

Return Offset instance.

RhythmTreeLeaf.**stop_offset**
The stopping offset of a node in a rhythm-tree relative the root.

RhythmTreeLeaf.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

RhythmTreeLeaf.**is_pitched**
True if leaf is pitched:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf()
>>> leaf.is_pitched
True
```

```
>>> leaf.is_pitched = False
>>> leaf.is_pitched
False
```

Return boolean.

`RhythmTreeLeaf.`**name**

`RhythmTreeLeaf.`**preprolated_duration**

The node's preprolated_duration in pulses:

```
>>> node = rhythmtreetools.RhythmTreeLeaf(preprolated_duration=1)
>>> node.preprolated_duration
Duration(1, 1)
```

```
>>> node.preprolated_duration = 2
>>> node.preprolated_duration
Duration(2, 1)
```

Return int.

Special methods

`RhythmTreeLeaf.`**__call__**(*pulse_duration*)

Generate Abjad score components:

```
>>> leaf = rhythmtreetools.RhythmTreeLeaf(5)
>>> leaf((1, 4))
[Note("c'1"), Note("c'4")]
```

Return sequence of components.

`RhythmTreeLeaf.`**__copy__**(**args*)

`RhythmTreeLeaf.`**__deepcopy__**(**args*)

`RhythmTreeLeaf.`**__eq__**(*expr*)

`RhythmTreeLeaf.`**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`RhythmTreeLeaf.`**__getstate__**()

`RhythmTreeLeaf.`**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`RhythmTreeLeaf.`**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`RhythmTreeLeaf.`**__lt__**(*expr*)

Abjad objects by default do not implement this method.

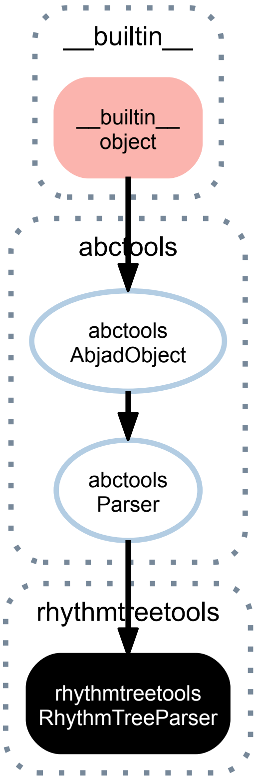
Raise exception.

`RhythmTreeLeaf.`**__ne__**(*expr*)

`RhythmTreeLeaf.`**__repr__**()

`RhythmTreeLeaf.`**__setstate__**(*state*)

25.2.3 rhythmtreetools.RhythmTreeParser



class rhythmtreetools.**RhythmTreeParser**(*debug=False*)

Parses RTM-style rhythm syntax:

```
>>> parser = rhythmtreetools.RhythmTreeParser()
```

```
>>> rtm = '(1 (1 (2 (1 -1 1)) -2))'
>>> result = parser(rtm)[0]
>>> result
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeContainer(
            children=(
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 1),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 1),
                    is_pitched=False
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 1),
                    is_pitched=True
                    )
                ),
            preprolated_duration=Duration(2, 1)
            ),
        RhythmTreeLeaf(
            preprolated_duration=Duration(2, 1),
            is_pitched=False
            )
        ),
    preprolated_duration=Duration(1, 1)
    )
```

```
>>> result.rtm_format
'(1 (1 (2 (1 -1 1)) -2))'
```

Returns *RhythmTreeParser* instance.

Read-only properties

RhythmTreeParser.**debug**
True if the parser runs in debugging mode.

RhythmTreeParser.**lexer**
The parser's PLY Lexer instance.

RhythmTreeParser.**lexer_rules_object**

RhythmTreeParser.**logger**
The parser's Logger instance.

RhythmTreeParser.**logger_path**
The output path for the parser's logfile.

RhythmTreeParser.**output_path**
The output path for files associated with the parser.

RhythmTreeParser.**parser**
The parser's PLY LRParser instance.

RhythmTreeParser.**parser_rules_object**

RhythmTreeParser.**pickle_path**
The output path for the parser's pickled parsing tables.

RhythmTreeParser.**storage_format**
Storage format of Abjad object.

Return string.

Methods

RhythmTreeParser.**p_container__LPAREN__DURATION__node_list_closed__RPAREN**(*p*)
container : LPAREN DURATION node_list_closed RPAREN

RhythmTreeParser.**p_error**(*p*)

RhythmTreeParser.**p_leaf__INTEGER**(*p*)
leaf : DURATION

RhythmTreeParser.**p_node__container**(*p*)
node : container

RhythmTreeParser.**p_node__leaf**(*p*)
node : leaf

RhythmTreeParser.**p_node_list__node_list__node_list_item**(*p*)
node_list : node_list node_list_item

RhythmTreeParser.**p_node_list__node_list_item**(*p*)
node_list : node_list_item

RhythmTreeParser.**p_node_list_closed__LPAREN__node_list__RPAREN**(*p*)
node_list_closed : LPAREN node_list RPAREN

RhythmTreeParser.**p_node_list_item__node**(*p*)
node_list_item : node

`RhythmTreeParser.`**`p_toplevel__EMPTY`**`(`*p*`)`
toplevel :

`RhythmTreeParser.`**`p_toplevel__toplevel__node`**`(`*p*`)`
toplevel : toplevel node

`RhythmTreeParser.`**`t_DURATION`**`(`*t*`)`
-?[1-9]d*(/[1-9]d*)?

`RhythmTreeParser.`**`t_error`**`(`*t*`)`

`RhythmTreeParser.`**`t_newline`**`(`*t*`)`
n+

`RhythmTreeParser.`**`tokenize`**`(`*input_string*`)`
Tokenize *input string* and print results.

Special methods

`RhythmTreeParser.`**`__call__`**`(`*input_string*`)`
Parse *input_string* and return result.

`RhythmTreeParser.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`RhythmTreeParser.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`RhythmTreeParser.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`RhythmTreeParser.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`RhythmTreeParser.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`RhythmTreeParser.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`RhythmTreeParser.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

25.3 Functions

25.3.1 rhythmtreetools.parse_rtm_syntax

`rhythmtreetools.`**`parse_rtm_syntax`**`(`*rtm*`)`
Parse RTM syntax:

```
>>> rtm = '(1 (1 (1 (1 1)) 1))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(1/4, [c'8, c'16, c'16, c'8])
```

Also supports fractional durations:

```
>>> rtm = '(3/4 (1 1/2 (4/3 (1 -1/2 1))))'
>>> rhythmtreetools.parse_rtm_syntax(rtm)
FixedDurationTuplet(3/16, [c'8, c'16, {@ 15:8 c'8, r16, c'8 @}])
>>> f(_)
\fraction \times 9/17 {
    c'8
    c'16
    \times 8/15 {
        c'8
        r16
        c'8
    }
}
```

Return *FixedDurationTuplet* or *Container* instance.

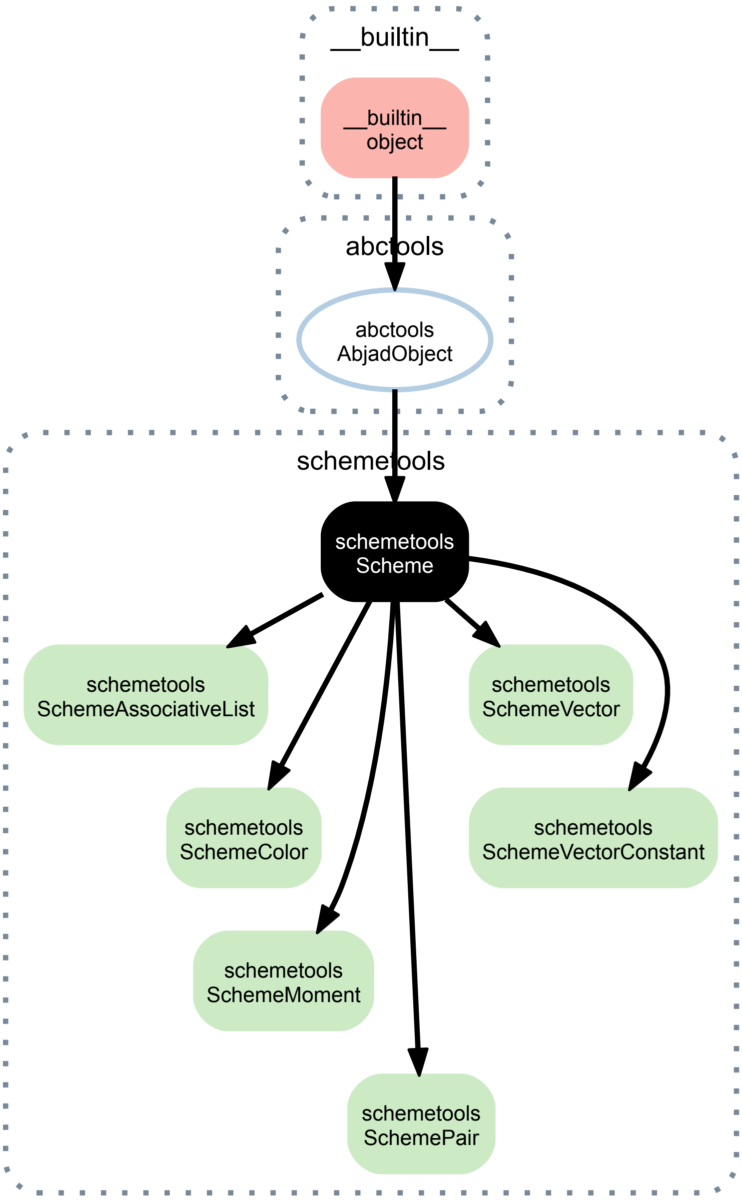
CHAPTER

TWENTYSIX

SCHEMETOOLS

26.1 Concrete Classes

26.1.1 schemetools.Scheme



class schemetools.**Scheme**(**args*, ***kwargs*)

Abjad model of Scheme code:

```
>>> s = schemetools.Scheme(True)
>>> s.lilypond_format
'##t'
```

schemetools.Scheme can represent nested structures:

```
>>> s = schemetools.Scheme(('left', (1, 2, False)), ('right', (1, 2, 3.3)))
>>> s.lilypond_format
'#((left (1 2 #f)) (right (1 2 3.3)))'
```

schemetools.Scheme wraps variable-length arguments into a tuple:

```
>>> s = schemetools.Scheme(1, 2, 3)
>>> q = schemetools.Scheme((1, 2, 3))
>>> s.lilypond_format == q.lilypond_format
True
```

schemetools.Scheme also takes an optional *quoting* keyword, by which Scheme's various quote, unquote, unquote-splicing characters can be prepended to the formatted result:

```
>>> s = schemetools.Scheme((1, 2, 3), quoting="'#")
>>> s.lilypond_format
"#'#(1 2 3)"
```

schemetools.Scheme can also force quotes around strings which contain no whitespace:

```
>>> s = schemetools.Scheme('nospaces', force_quotes=True)
>>> f(s)
#"nospaces"
```

The above is useful in certain override situations, as LilyPond's Scheme interpreter will treat unquoted strings as symbols rather than strings.

Scheme is immutable.

Read-only properties

Scheme.**force_quotes**

Scheme.**lilypond_format**

Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

Scheme.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

Scheme.**__eq__**(*expr*)

Scheme.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Scheme.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

Scheme.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Scheme.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

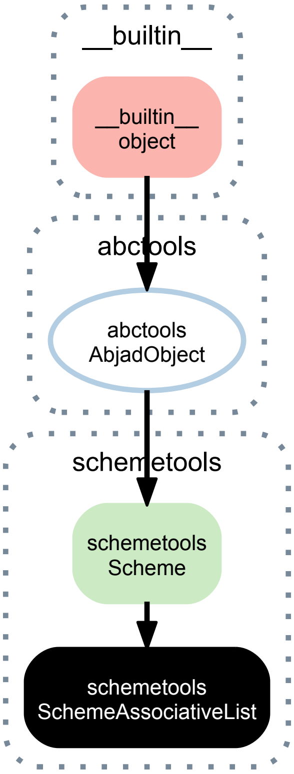
Scheme.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Scheme.**__repr__**()

Scheme.**__str__**()

26.1.2 schemetools.SchemeAssociativeList



class schemetools.**SchemeAssociativeList**(**args*, ***kwargs*)
New in version 2.0. Abjad model of Scheme associative list:

```
>>> schemetools.SchemeAssociativeList(('space', 2), ('padding', 0.5))
SchemeAssociativeList((SchemePair(('space', 2)), SchemePair(('padding', 0.5))))
```

Scheme associative lists are immutable.

Read-only properties

SchemeAssociativeList.**force_quotes**

SchemeAssociativeList.**lilypond_format**
Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

SchemeAssociativeList.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

SchemeAssociativeList.**__eq__**(*expr*)

SchemeAssociativeList.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemeAssociativeList.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

SchemeAssociativeList.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemeAssociativeList.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemeAssociativeList.**__ne__**(*expr*)

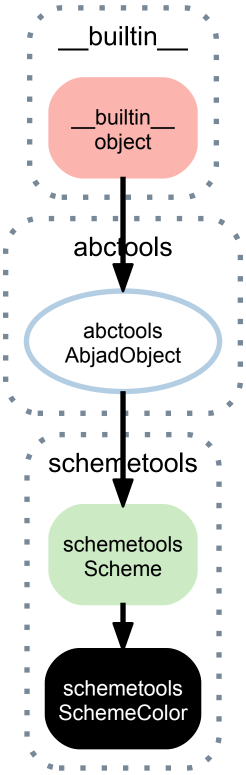
Defined equal to the opposite of equality.

Return boolean.

SchemeAssociativeList.**__repr__**()

SchemeAssociativeList.**__str__**()

26.1.3 schemetools.SchemeColor



class schemetools.**SchemeColor**(**args*, ***kwargs*)

Abjad model of Scheme color:

```
>>> schemetools.SchemeColor('ForestGreen')
SchemeColor('ForestGreen')
```

Scheme colors are immutable.

Read-only properties

SchemeColor.**force_quotes**

SchemeColor.**lilypond_format**

Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

SchemeColor.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

SchemeColor.**__eq__**(*expr*)

SchemeColor.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemeColor.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SchemeColor.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeColor.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

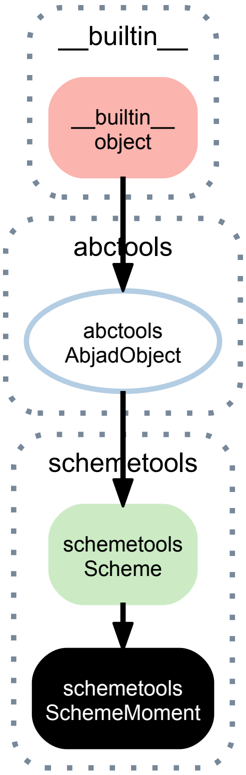
SchemeColor.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SchemeColor.**__repr__**()

SchemeColor.**__str__**()

26.1.4 schemetools.SchemeMoment



class schemetools.**SchemeMoment**(**args*, ***kwargs*)
Abjad model of LilyPond moment:

```
>>> schemetools.SchemeMoment(1, 68)
SchemeMoment((1, 68))
```

Initialize scheme moments with a single fraction, two integers or another scheme moment.

Scheme moments are immutable.

Read-only properties

`SchemeMoment.`**`duration`**

Duration of scheme moment:

```
>>> scheme_moment = schemetools.SchemeMoment(1, 68)
>>> scheme_moment.duration
Duration(1, 68)
```

Return duration.

`SchemeMoment.`**`force_quotes`**

`SchemeMoment.`**`lilypond_format`**

Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

`SchemeMoment.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`SchemeMoment.`**`__eq__`**`(`*arg*`)`

`SchemeMoment.`**`__ge__`**`(`*arg*`)`

`SchemeMoment.`**`__gt__`**`(`*arg*`)`

`SchemeMoment.`**`__le__`**`(`*arg*`)`

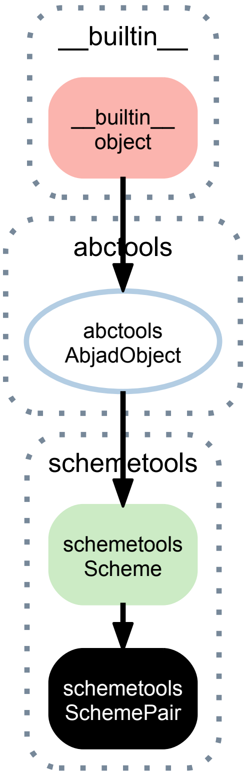
`SchemeMoment.`**`__lt__`**`(`*arg*`)`

`SchemeMoment.`**`__ne__`**`(`*arg*`)`

`SchemeMoment.`**`__repr__`**`()`

`SchemeMoment.`**`__str__`**`()`

26.1.5 schemetools.SchemePair



class schemetools.**SchemePair**(**args, **kwargs*)

Abjad model of Scheme pair:

```
>>> schemetools.SchemePair('spacing', 4)
SchemePair(('spacing', 4))
```

Initialize Scheme pairs with a tuple, two separate values or another Scheme pair.

Scheme pairs are immutable.

Read-only properties

SchemePair.**force_quotes**

SchemePair.**lilypond_format**

SchemePair.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

SchemePair.**__eq__**(*expr*)

SchemePair.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemePair.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

SchemePair.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemePair.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

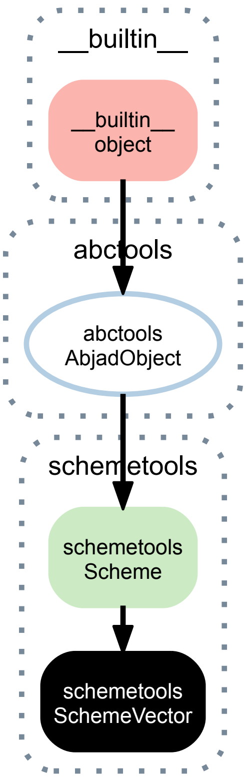
SchemePair.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SchemePair.**__repr__**()

SchemePair.**__str__**()

26.1.6 schemetools.SchemeVector



class schemetools.**SchemeVector**(**args*)
New in version 2.0. Abjad model of Scheme vector:

```
>>> schemetools.SchemeVector(True, True, False)
SchemeVector((True, True, False))
```

Scheme vectors and Scheme vector constants differ in only their LilyPond input format.

Scheme vectors are immutable.

Read-only properties

SchemeVector.**force_quotes**

SchemeVector.**lilypond_format**
Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

SchemeVector.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

SchemeVector.**__eq__**(*expr*)

SchemeVector.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeVector.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SchemeVector.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeVector.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

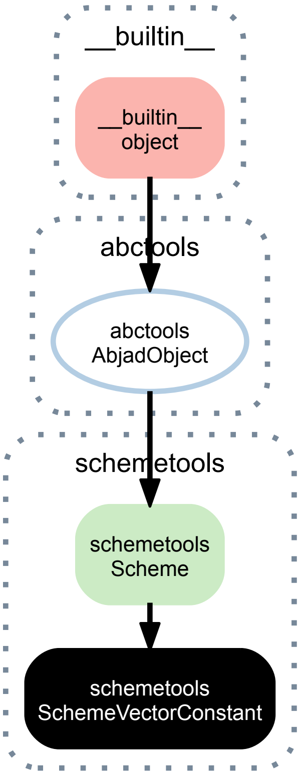
SchemeVector.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SchemeVector.**__repr__**()

SchemeVector.**__str__**()

26.1.7 schemetools.SchemeVectorConstant



class schemetools.**SchemeVectorConstant**(**args*)

New in version 2.0. Abjad model of Scheme vector constant:

```
>>> schemetools.SchemeVectorConstant(True, True, False)
SchemeVectorConstant((True, True, False))
```

Scheme vectors and Scheme vector constants differ in only their LilyPond input format.

Scheme vector constants are immutable.

Read-only properties

SchemeVectorConstant.**force_quotes**

SchemeVectorConstant.**lilypond_format**

Hash-mark-prepended format of Scheme:

```
>>> schemetools.Scheme(True).lilypond_format
'##t'
```

Returns string.

SchemeVectorConstant.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

SchemeVectorConstant.**__eq__**(*expr*)

SchemeVectorConstant.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SchemeVectorConstant.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SchemeVectorConstant.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeVectorConstant.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeVectorConstant.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SchemeVectorConstant.**__repr__**()

SchemeVectorConstant.**__str__**()

26.2 Functions

26.2.1 schemetools.format_scheme_value

schemetools.**format_scheme_value**(*value*, *force_quotes=False*)
Format *value* as Scheme would:

```
>>> schemetools.format_scheme_value(1)
'1'
```

```
>>> schemetools.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.format_scheme_value('bar baz')
'"bar baz"'
```

```
>>> schemetools.format_scheme_value([1.5, True, False])
'(1.5 #t #f)'
```

Strings without whitespace can be forcibly quoted via the *force_quotes* keyword:

```
>>> schemetools.format_scheme_value('foo', force_quotes=True)
'"foo"'
```

Return string.

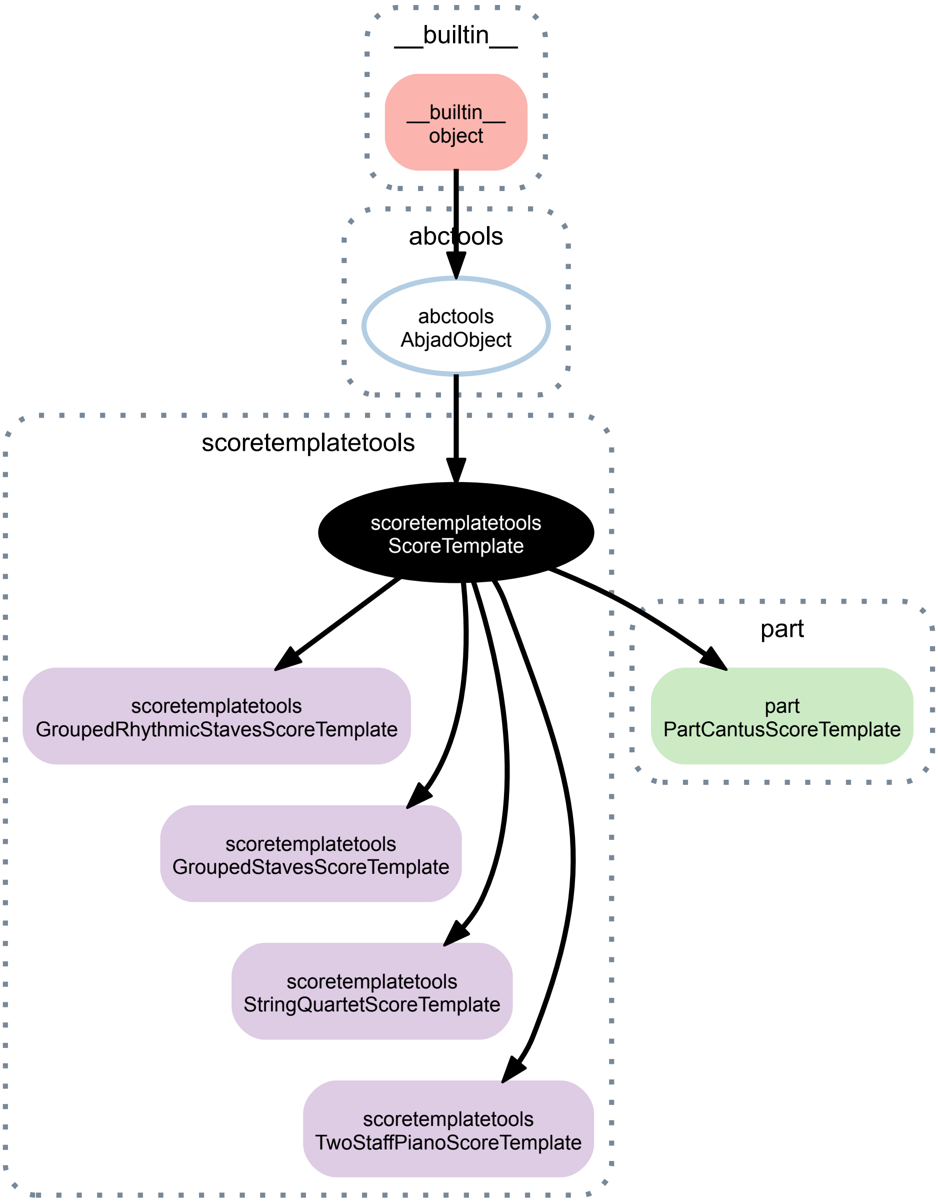
CHAPTER

TWENTYSEVEN

SCORETEMPLATETOOLS

27.1 Abstract Classes

27.1.1 scoretemplatetools.ScoreTemplate



class `scoretemplatetools.`**`ScoreTemplate`**

New in version 2.8. Abstract score template.

Read-only properties

`ScoreTemplate.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`ScoreTemplate.`**`__call__`**`()`

`ScoreTemplate.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`ScoreTemplate.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`ScoreTemplate.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`ScoreTemplate.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`ScoreTemplate.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`ScoreTemplate.`**`__ne__`**`(`*expr*`)`

Defined equal to the opposite of equality.

Return boolean.

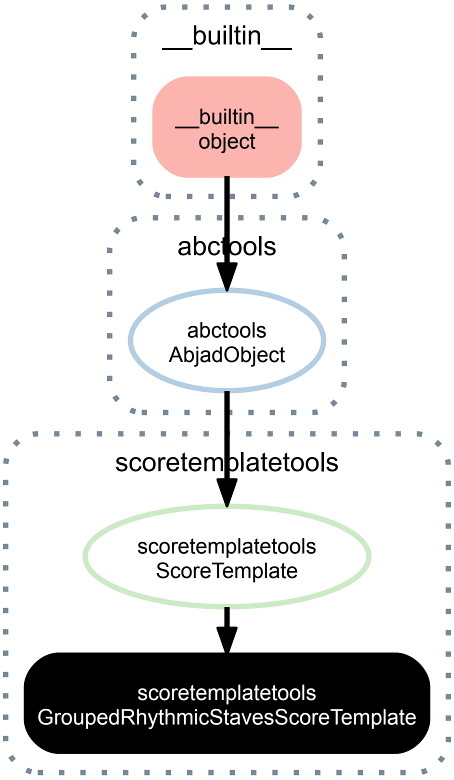
`ScoreTemplate.`**`__repr__`**`()`

Interpreter representation of Abjad object.

Return string.

27.2 Concrete Classes

27.2.1 scoretemplatetools.GroupedRhythmicStavesScoreTemplate



class scoretemplatetools.**GroupedRhythmicStavesScoreTemplate**(*staff_count=2*)

New in version 2.9. Example 1. Grouped rhythmic staves score template with one voice per staff:

```
>>> template = scoretemplatetools.GroupedRhythmicStavesScoreTemplate(staff_count=4)
>>> score = template()
```

```
>>> score
Score-"Grouped Rhythmic Staves Score"<<1>>
```

```
>>> f(score)
\context Score = "Grouped Rhythmic Staves Score" <<
    \context StaffGroup = "Grouped Rhythmic Staves Staff Group" <<
        \context RhythmicStaff = "Staff 1" {
            \context Voice = "Voice 1" {
            }
        }
        \context RhythmicStaff = "Staff 2" {
            \context Voice = "Voice 2" {
            }
        }
        \context RhythmicStaff = "Staff 3" {
            \context Voice = "Voice 3" {
            }
        }
        \context RhythmicStaff = "Staff 4" {
            \context Voice = "Voice 4" {
            }
        }
    >>
>>
```

Example 2. Grouped rhythmic staves score template with more than one voice per staff:

```
>>> template = scoretemplatetools.GroupedRhythmicStavesScoreTemplate(staff_count=[2, 1, 2])
>>> score = template()
```

```
>>> f(score)
\context Score = "Grouped Rhythmic Staves Score" <<
    \context StaffGroup = "Grouped Rhythmic Staves Staff Group" <<
        \context RhythmicStaff = "Staff 1" <<
            \context Voice = "Voice 1-1" {
            }
            \context Voice = "Voice 1-2" {
            }
        >>
        \context RhythmicStaff = "Staff 2" {
            \context Voice = "Voice 2" {
            }
        }
        \context RhythmicStaff = "Staff 3" <<
            \context Voice = "Voice 3-1" {
            }
            \context Voice = "Voice 3-2" {
            }
        >>
    >>
>>
```

Return score template object.

Read-only properties

GroupedRhythmicStavesScoreTemplate.**staff_count**

GroupedRhythmicStavesScoreTemplate.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

GroupedRhythmicStavesScoreTemplate.**__call__**()

GroupedRhythmicStavesScoreTemplate.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

GroupedRhythmicStavesScoreTemplate.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GroupedRhythmicStavesScoreTemplate.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GroupedRhythmicStavesScoreTemplate.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GroupedRhythmicStavesScoreTemplate.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

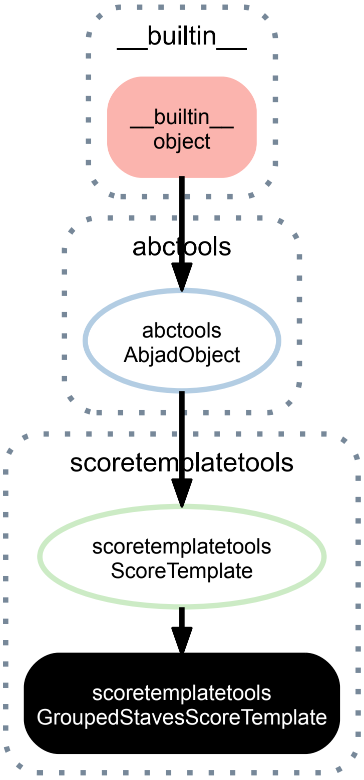
GroupedRhythmicStavesScoreTemplate.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

GroupedRhythmicStavesScoreTemplate.**__repr__**()
Interpreter representation of Abjad object.

Return string.

27.2.2 scoretemplatetools.GroupedStavesScoreTemplate



class scoretemplatetools.**GroupedStavesScoreTemplate**(*staff_count=2*)
New in version 2.10. Grouped staves score template:

```
>>> template = scoretemplatetools.GroupedStavesScoreTemplate(staff_count=4)
>>> score = template()
```

```
>>> score
Score-"Grouped Staves Score"<<1>>
```

```
>>> f(score)
\context Score = "Grouped Staves Score" <<
    \context StaffGroup = "Grouped Staves Staff Group" <<
        \context Staff = "Staff 1" {
            \context Voice = "Voice 1" {
            }
        }
        \context Staff = "Staff 2" {
            \context Voice = "Voice 2" {
            }
        }
        \context Staff = "Staff 3" {
            \context Voice = "Voice 3" {
            }
        }
        \context Staff = "Staff 4" {
            \context Voice = "Voice 4" {
            }
        }
```

```
        >>
    >>
```

Return score template object.

Read-only properties

GroupedStavesScoreTemplate.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

GroupedStavesScoreTemplate.**__call__**()

GroupedStavesScoreTemplate.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

GroupedStavesScoreTemplate.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

GroupedStavesScoreTemplate.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

GroupedStavesScoreTemplate.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

GroupedStavesScoreTemplate.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

GroupedStavesScoreTemplate.**__ne__**(*expr*)

Defined equal to the opposite of equality.

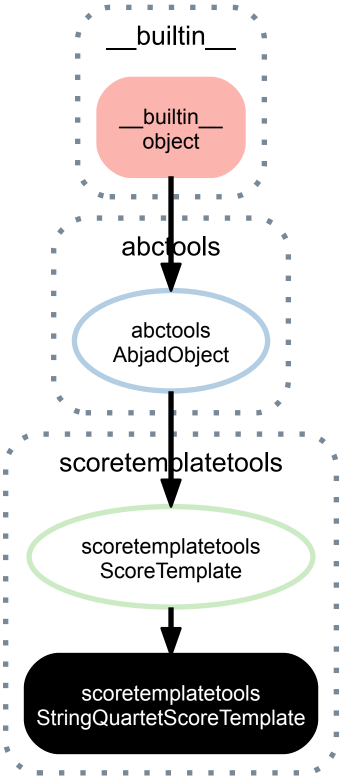
Return boolean.

GroupedStavesScoreTemplate.**__repr__**()

Interpreter representation of Abjad object.

Return string.

27.2.3 scoretemplatetools.StringQuartetScoreTemplate



class scoretemplatetools.**StringQuartetScoreTemplate**

New in version 2.8. String quartet score template:

```
>>> template = scoretemplatetools.StringQuartetScoreTemplate()
>>> score = template()
```

```
>>> score
Score-"String Quartet Score"<<1>>
```

```
>>> f(score)
\context Score = "String Quartet Score" <<
    \context StaffGroup = "String Quartet Staff Group" <<
        \context Staff = "First Violin Staff" {
            \clef "treble"
            \set Staff.instrumentName = \markup { Violin }
            \set Staff.shortInstrumentName = \markup { Vn. }
            \context Voice = "First Violin Voice" {
            }
        }
        \context Staff = "Second Violin Staff" {
            \clef "treble"
            \set Staff.instrumentName = \markup { Violin }
            \set Staff.shortInstrumentName = \markup { Vn. }
            \context Voice = "Second Violin Voice" {
            }
        }
        \context Staff = "Viola Staff" {
            \clef "alto"
            \set Staff.instrumentName = \markup { Viola }
            \set Staff.shortInstrumentName = \markup { Va. }
            \context Voice = "Viola Voice" {
            }
        }
        \context Staff = "Cello Staff" {
            \clef "bass"
            \set Staff.instrumentName = \markup { Cello }
            \set Staff.shortInstrumentName = \markup { Vc. }
            \context Voice = "Cello Voice" {
```

```
                }
            }
        >>
    >>
```

Return score template.

Read-only properties

StringQuartetScoreTemplate.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

StringQuartetScoreTemplate.**__call__**()

StringQuartetScoreTemplate.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

StringQuartetScoreTemplate.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StringQuartetScoreTemplate.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

StringQuartetScoreTemplate.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StringQuartetScoreTemplate.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

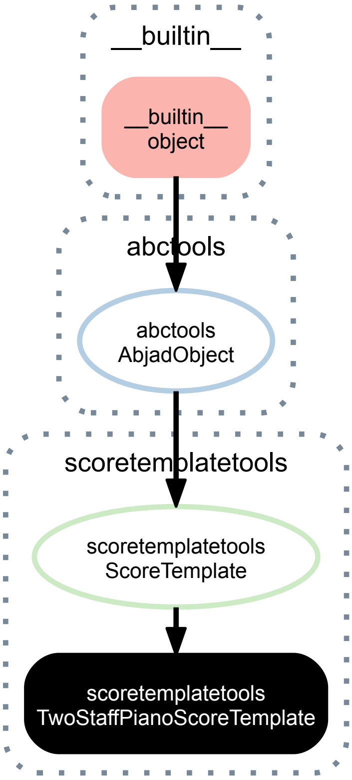
StringQuartetScoreTemplate.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

StringQuartetScoreTemplate.**__repr__**()
Interpreter representation of Abjad object.

Return string.

27.2.4 scoretemplatetools.TwoStaffPianoScoreTemplate



class scoretemplatetools.**TwoStaffPianoScoreTemplate**

New in version 2.8. Two-staff piano score template:

```
>>> template = scoretemplatetools.TwoStaffPianoScoreTemplate()
>>> score = template()
```

```
>>> score
Score-"Two-Staff Piano Score"<<1>>
```

```
>>> f(score)
\context Score = "Two-Staff Piano Score" <<
    \context PianoStaff = "Piano Staff" <<
        \set PianoStaff.instrumentName = \markup { Piano }
        \set PianoStaff.shortInstrumentName = \markup { Pf. }
        \context Staff = "RH Staff" {
            \clef "treble"
            \context Voice = "RH Voice" {
            }
        }
        \context Staff = "LH Staff" {
            \clef "bass"
            \context Voice = "LH Voice" {
            }
        }
    >>
>>
```

Return score template.

Read-only properties

TwoStaffPianoScoreTemplate.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

`TwoStaffPianoScoreTemplate.__call__()`

`TwoStaffPianoScoreTemplate.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`TwoStaffPianoScoreTemplate.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TwoStaffPianoScoreTemplate.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TwoStaffPianoScoreTemplate.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TwoStaffPianoScoreTemplate.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TwoStaffPianoScoreTemplate.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`TwoStaffPianoScoreTemplate.__repr__()`
Interpreter representation of Abjad object.

Return string.

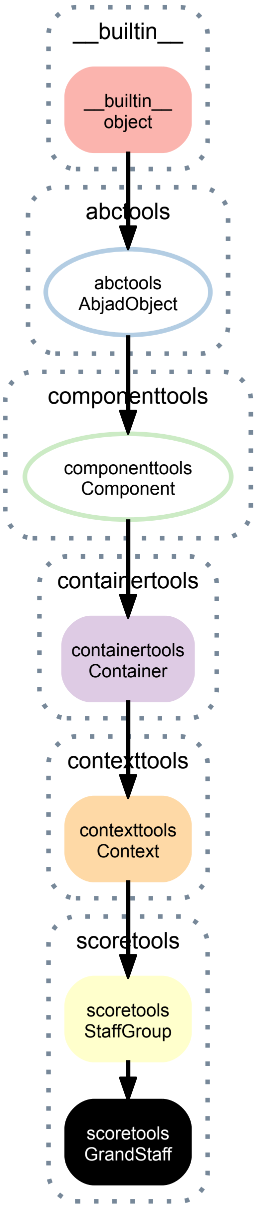
CHAPTER

TWENTYEIGHT

SCORETOOLS

28.1 Concrete Classes

28.1.1 scoretools.GrandStaff



class `scoretools.`**`GrandStaff`**(*music*, *context_name='GrandStaff'*, *name=None*)

Abjad model of grand staff:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")
```

```
>>> grand_staff = scoretools.GrandStaff([staff_1, staff_2])
```

```
>>> f(grand_staff)
\new GrandStaff <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
        g'1
    }
    \new Staff {
        g2
        f2
        e1
    }
>>
```

Return grand staff.

Read-only properties

`GrandStaff.`**`contents_duration`**

`GrandStaff.`**`descendants`**

Read-only reference to component descendants score selection.

`GrandStaff.`**`duration`**

`GrandStaff.`**`duration_in_seconds`**

`GrandStaff.`**`engraver_consists`**

New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

`GrandStaff.`**`engraver_removals`**

New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

`GrandStaff.`**`is_semantic`**

`GrandStaff.`**`leaves`**

Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`GrandStaff.`**`lilypond_format`**

`GrandStaff.`**`lineage`**
Read-only reference to component lineage score selection.

`GrandStaff.`**`music`**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`GrandStaff.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`GrandStaff.`**`parent`**

`GrandStaff.`**`parentage`**
Read-only reference to component parentage score selection.

`GrandStaff.`**`preprolated_duration`**

`GrandStaff.`**`prolation`**

`GrandStaff.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`GrandStaff.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`GrandStaff.`**`storage_format`**
Storage format of Abjad object.

Return string.

`GrandStaff.`**`timespan`**
Read-only timespan of component.

`GrandStaff.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Read/write properties

`GrandStaff.`**`context_name`**
Read / write name of context as a string.

`GrandStaff.`**`is_nonsemantic`**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

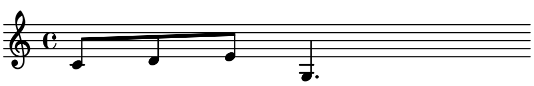
GrandStaff.**is_parallel**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

Return boolean.

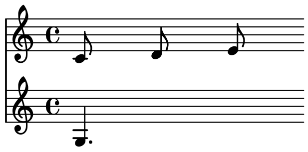
Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
```

```
        \new Voice {
            g4.
        }
    >>
```

```
>>> show(container)
```



Return none.

GrandStaff.**name**
Read-write name of context. Must be string or none.

Methods

GrandStaff.**append**(*component*)
Append *component* to container:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

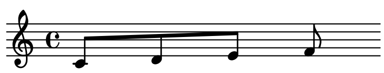
```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f’8"))
```

```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
    f’8
}
```

```
>>> show(container)
```



Return none.

GrandStaff.**extend**(*expr*)
Extend *expr* against container:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

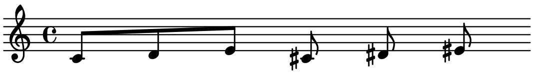
```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`GrandStaff.`**`index`**(*component*)
Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`GrandStaff.`**`insert`**(*i*, *component*)
Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

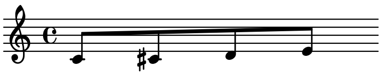
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`GrandStaff.pop(i=-1)`
Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

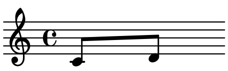
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`GrandStaff.remove(component)`
Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

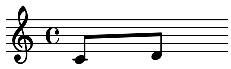
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

GrandStaff.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

GrandStaff.**__contains__**(*expr*)
True if expr is in container, otherwise False.

GrandStaff.**__copy__**(**args*)

GrandStaff.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

GrandStaff.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

GrandStaff.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GrandStaff.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

GrandStaff.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GrandStaff.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

GrandStaff.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

GrandStaff.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GrandStaff.**__len__**()
Return nonnegative integer number of components in container.

GrandStaff.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GrandStaff.**__mul__**(*n*)

GrandStaff.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

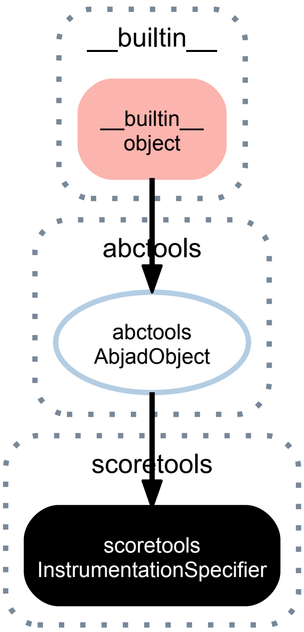
GrandStaff.**__radd__**(*expr*)
Extend container by contents of expr to the right.

GrandStaff.**__repr__**()

GrandStaff.**__rmul__**(*n*)

GrandStaff.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

28.1.2 scoretools.InstrumentationSpecifier



class scoretools.**InstrumentationSpecifier**(*performers=None*)
New in version 2.5. Abjad model of score instrumentation:

```
>>> flute = scoretools.Performer('Flute')
>>> flute.instruments.append(instrumenttools.Flute())
>>> flute.instruments.append(instrumenttools.AltoFlute())
```

```
>>> guitar = scoretools.Performer('Guitar')
>>> guitar.instruments.append(instrumenttools.Guitar())
```

```
>>> instrumentation_specifier = scoretools.InstrumentationSpecifier([flute, guitar])
```

```
>>> z(instrumentation_specifier)
scoretools.InstrumentationSpecifier(
    performers=scoretools.PerformerInventory([
        scoretools.Performer(
            name='Flute',
            instruments=instrumenttools.InstrumentInventory([
                instrumenttools.Flute(),
                instrumenttools.AltoFlute()
                ])
            ),
        scoretools.Performer(
            name='Guitar',
```

```
                instruments=instrumenttools.InstrumentInventory([
                    instrumenttools.Guitar()
                    ])
                )
            ])
        )
```

Return instrumentation specifier.

Read-only properties

`InstrumentationSpecifier.`**`instrument_count`**

Read-only number of instruments in score:

```
>>> instrumentation_specifier.instrument_count
3
```

Return nonnegative integer.

`InstrumentationSpecifier.`**`instruments`**

Read-only list of instruments derived from performers:

```
>>> instrumentation_specifier.instruments
[Flute(), AltoFlute(), Guitar()]
```

Return list.

`InstrumentationSpecifier.`**`performer_count`**

Read-only number of performers in score:

```
>>> instrumentation_specifier.performer_count
2
```

Return nonnegative integer.

`InstrumentationSpecifier.`**`performer_name_string`**

Read-only string of performer names:

```
>>> instrumentation_specifier.performer_name_string
'Flute, Guitar'
```

Return string.

`InstrumentationSpecifier.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`InstrumentationSpecifier.`**`performers`**

Read / write list of performers in score:

```
>>> z(instrumentation_specifier.performers)
scoretools.PerformerInventory([
    scoretools.Performer(
        name='Flute',
        instruments=instrumenttools.InstrumentInventory([
            instrumenttools.Flute(),
            instrumenttools.AltoFlute()
            ])
        ),
    scoretools.Performer(
        name='Guitar',
        instruments=instrumenttools.InstrumentInventory([
            instrumenttools.Guitar()
            ])
```

```
            )
        ])
```

Return performer inventory.

Special methods

InstrumentationSpecifier.**__eq__**(*expr*)

InstrumentationSpecifier.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InstrumentationSpecifier.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

InstrumentationSpecifier.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InstrumentationSpecifier.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

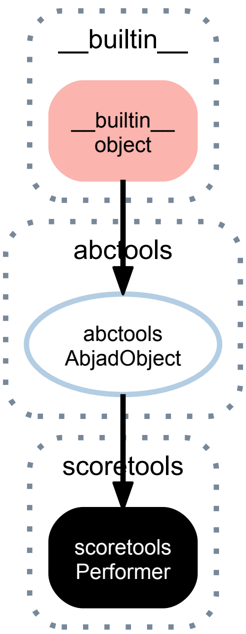
Raise exception.

InstrumentationSpecifier.**__ne__**(*expr*)

InstrumentationSpecifier.**__repr__**()
Interpreter representation of Abjad object.

Return string.

28.1.3 scoretools.Performer



class scoretools.**Performer**(*name=None*, *instruments=None*)
New in version 2.5. Abjad model of performer:

```
>>> scoretools.Performer(name='flutist')
Performer(name='flutist', instruments=InstrumentInventory([]))
```

The purpose of the class is to model things like flute I doubling piccolo and alto flute.

At present the class is a list of instruments.

Read-only properties

`Performer.`**instrument_count**

Read-only number of instruments to be played by performer:

```
>>> performer = scoretools.Performer('flutist')
```

```
>>> performer.instruments.append(instrumenttools.Flute())
>>> performer.instruments.append(instrumenttools.Piccolo())
```

```
>>> performer.instrument_count
2
```

Return nonnegative integer

`Performer.`**is_doubling**

Is performer to play more than one instrument?

```
>>> performer = scoretools.Performer('flutist')
```

```
>>> performer.instruments.append(instrumenttools.Flute())
>>> performer.instruments.append(instrumenttools.Piccolo())
```

```
>>> performer.is_doubling
True
```

Return boolean.

`Performer.`**likely_instruments_based_on_performer_name**

New in version 2.5. Likely instruments based on performer name:

```
>>> flutist = scoretools.Performer(name='flutist')
>>> for likely_instrument in flutist.likely_instruments_based_on_performer_name:
...     likely_instrument
...
<class 'abjad.tools.instrumenttools.AltoFlute.AltoFlute.AltoFlute'>
<class 'abjad.tools.instrumenttools.BassFlute.BassFlute.BassFlute'>
<class 'abjad.tools.instrumenttools.ContrabassFlute.ContrabassFlute.ContrabassFlute'>
<class 'abjad.tools.instrumenttools.Flute.Flute.Flute'>
<class 'abjad.tools.instrumenttools.Piccolo.Piccolo.Piccolo'>
```

Return list.

`Performer.`**most_likely_instrument_based_on_performer_name**

New in version 2.5. Most likely instrument based on performer name:

```
>>> flutist = scoretools.Performer(name='flutist')
>>> flutist.most_likely_instrument_based_on_performer_name
<class 'abjad.tools.instrumenttools.Flute.Flute.Flute'>
```

Return instrument class.

`Performer.`**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

`Performer.`**`instruments`**
List of instruments to be played by performer:

```
>>> performer = scoretools.Performer('flutist')
```

```
>>> performer.instruments.append(instrumenttools.Flute())
>>> performer.instruments.append(instrumenttools.Piccolo())
```

```
>>> performer.instruments
InstrumentInventory([Flute(), Piccolo()])
```

Return list.

`Performer.`**`name`**
Score name of performer:

```
>>> performer = scoretools.Performer('flutist')
```

```
>>> performer.name
'flutist'
```

Return string.

Methods

`Performer.`**`make_performer_name_instrument_dictionary`**(*locale=None*)
New in version 2.5. Make performer name / instrument dictionary.

Return dictionary.

Special methods

`Performer.`**`__eq__`**(*expr*)

`Performer.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Performer.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`Performer.`**`__hash__`**()

`Performer.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`Performer.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

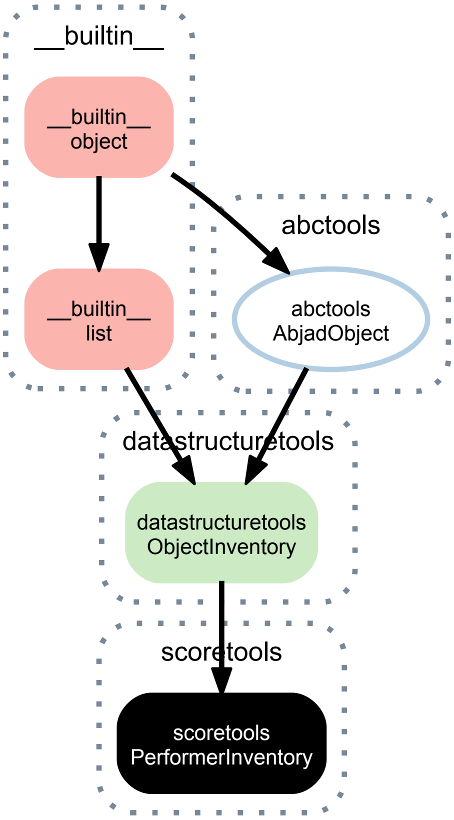
Raise exception.

`Performer.`**`__ne__`**(*expr*)

`Performer.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

28.1.4 scoretools.PerformerInventory



class `scoretools.`**`PerformerInventory`**(*tokens=None*, *name=None*)
New in version 2.8. Abjad model of an ordered list of performers.

Performer inventories implement the list interface and are mutable.

Read-only properties

`PerformerInventory.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`PerformerInventory.`**`name`**
Read / write name of inventory.

Methods

`PerformerInventory.`**`append`**(*token*)
Change *token* to item and append.

`PerformerInventory.`**`count`**(*value*) → integer – return number of occurrences of value

`PerformerInventory.`**`extend`**(*tokens*)
Change *tokens* to items and extend.

`PerformerInventory.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

PerformerInventory.**insert**()
L.insert(index, object) – insert object before index

PerformerInventory.**pop**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

PerformerInventory.**remove**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

PerformerInventory.**reverse**()
L.reverse() – reverse *IN PLACE*

PerformerInventory.**sort**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

PerformerInventory.**__add__**()
x.__add__(y) <==> x+y

PerformerInventory.**__contains__**(*token*)

PerformerInventory.**__delitem__**()
x.__delitem__(y) <==> del x[y]

PerformerInventory.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

PerformerInventory.**__eq__**()
x.__eq__(y) <==> x==y

PerformerInventory.**__ge__**()
x.__ge__(y) <==> x>=y

PerformerInventory.**__getitem__**()
x.__getitem__(y) <==> x[y]

PerformerInventory.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

PerformerInventory.**__gt__**()
x.__gt__(y) <==> x>y

PerformerInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

PerformerInventory.**__imul__**()
x.__imul__(y) <==> x*=y

PerformerInventory.**__iter__**() <==> iter(x)

PerformerInventory.**__le__**()
x.__le__(y) <==> x<=y

PerformerInventory.**__len__**() <==> len(x)

PerformerInventory.**__lt__**()
x.__lt__(y) <==> x<y

PerformerInventory.**__mul__**()
x.__mul__(n) <==> x*n

PerformerInventory.**__ne__**()
x.__ne__(y) <==> x!=y

`PerformerInventory.__repr__()`

`PerformerInventory.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

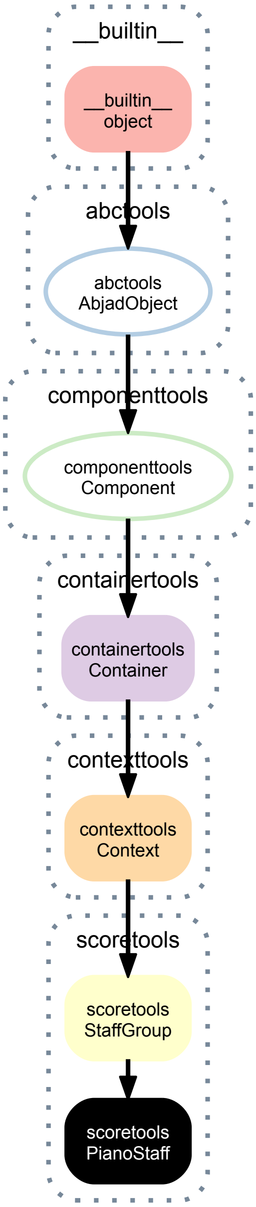
`PerformerInventory.__rmul__()`
x.__rmul__(n) <==> n*x

`PerformerInventory.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`PerformerInventory.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

28.1.5 scoretools.PianoStaff



class `scoretools.PianoStaff` (*music=None*, *context_name='PianoStaff'*, *name=None*)
Abjad model of piano staff:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")
```

```
>>> piano_staff = scoretools.PianoStaff([staff_1, staff_2])
```

```
>>> f(piano_staff)
\new PianoStaff <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
        g'1
    }
    \new Staff {
        g2
        f2
        e1
    }
>>
```

Return piano staff.

Read-only properties

PianoStaff.**contents_duration**

PianoStaff.**descendants**
Read-only reference to component descendants score selection.

PianoStaff.**duration**

PianoStaff.**duration_in_seconds**

PianoStaff.**engraver_consists**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

PianoStaff.**engraver_removals**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

PianoStaff.**is_semantic**

PianoStaff.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

PianoStaff.**lilypond_format**

PianoStaff.**lineage**
Read-only reference to component lineage score selection.

PianoStaff.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

PianoStaff.**override**
Read-only reference to LilyPond grob override component plug-in.

PianoStaff.**parent**

PianoStaff.**parentage**
Read-only reference to component parentage score selection.

PianoStaff.**preprolated_duration**

PianoStaff.**prolation**

PianoStaff.**set**
Read-only reference LilyPond context setting component plug-in.

PianoStaff.**spanners**
Read-only reference to unordered set of spanners attached to component.

PianoStaff.**storage_format**
Storage format of Abjad object.

Return string.

PianoStaff.**timespan**
Read-only timespan of component.

PianoStaff.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

PianoStaff.**context_name**
Read / write name of context as a string.

PianoStaff.**is_nonsemantic**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
```

```
            s1 * 5/16
        }
        {
            s1 * 5/16
        }
    }
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

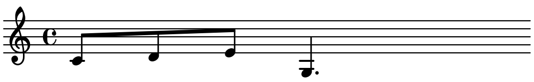
PianoStaff.**is_parallel**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

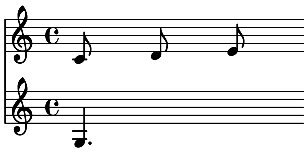
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

`PianoStaff.name`
Read-write name of context. Must be string or none.

Methods

`PianoStaff.append(component)`
Append *component* to container:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f’8"))
```

```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
    f’8
}
```

```
>>> show(container)
```

Return none.

`PianoStaff.extend(expr)`
Extend *expr* against container:

```
>>> container = Container("c’8 d’8 e’8")
>>> beam = beamtools.BeamSpanner(container.music)
```

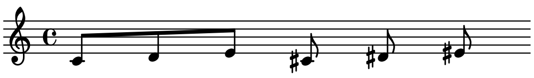
```
>>> f(container)
{
    c’8 [
    d’8
    e’8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`PianoStaff.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`PianoStaff.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

Return none.

PianoStaff.**pop**(*i=-1*)
Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

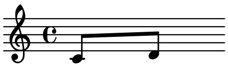
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

PianoStaff.**remove**(*component*)
Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

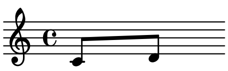
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

PianoStaff.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

PianoStaff.**__contains__**(*expr*)
True if expr is in container, otherwise False.

PianoStaff.**__copy__**(**args*)

PianoStaff.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

PianoStaff.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

PianoStaff.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PianoStaff.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

PianoStaff.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PianoStaff.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

PianoStaff.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

PianoStaff.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PianoStaff.**__len__**()
Return nonnegative integer number of components in container.

PianoStaff.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PianoStaff.**__mul__**(*n*)

PianoStaff.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

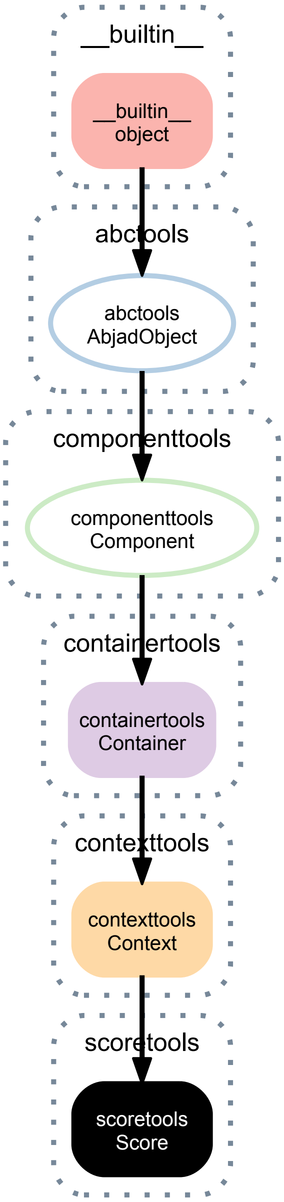
PianoStaff.**__radd__**(*expr*)
Extend container by contents of expr to the right.

PianoStaff.**__repr__**()

PianoStaff.**__rmul__**(*n*)

PianoStaff.**__setitem__**(*i*, *expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

28.1.6 scoretools.Score



class scoretools.**Score**(*music=None*, *context_name='Score'*, *name=None*)
Abjad model of a score:

```
>>> staff_1 = Staff("c'8 d'8 e'8 f'8")
>>> staff_2 = Staff("c'8 d'8 e'8 f'8")
>>> score = Score([staff_1, staff_2])
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
```

```
        \new Staff {
            c'8
            d'8
            e'8
            f'8
        }
    >>
```

Return Score instance.

Read-only properties

Score.**contents_duration**

Score.**descendants**
Read-only reference to component descendants score selection.

Score.**duration**

Score.**duration_in_seconds**

Score.**engraver_consists**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

Score.**engraver_removals**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

Score.**is_semantic**

Score.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

Score.**lilypond_format**

Score.**lineage**
Read-only reference to component lineage score selection.

Score.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`Score.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Score.`**`parent`**

`Score.`**`parentage`**
Read-only reference to component parentage score selection.

`Score.`**`preprolated_duration`**

`Score.`**`prolation`**

`Score.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Score.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`Score.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Score.`**`timespan`**
Read-only timespan of component.

`Score.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Read/write properties

`Score.`**`context_name`**
Read / write name of context as a string.

`Score.`**`is_nonsemantic`**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

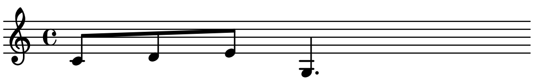
The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

Score.**is_parallel**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

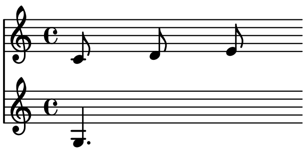
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

Score.**name**
Read-write name of context. Must be string or none.

Methods

`Score.append(component)`

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

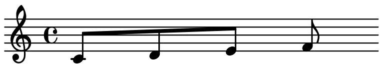
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`Score.extend(expr)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

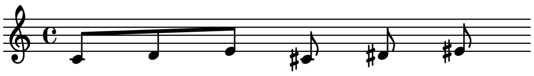
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Score.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Score.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

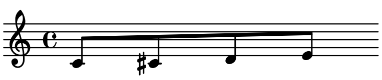
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Score.`**`pop`**(*i=-1*)

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

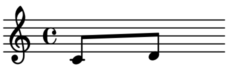
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Score.`**`remove`**(*component*)

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

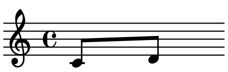
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Score.`**`__add__`**(*expr*)

Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

Score.**__contains__**(*expr*)
True if expr is in container, otherwise False.

Score.**__copy__**(**args*)

Score.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Score.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Score.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Score.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

Score.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Score.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

Score.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

Score.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Score.**__len__**()
Return nonnegative integer number of components in container.

Score.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Score.**__mul__**(*n*)

Score.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

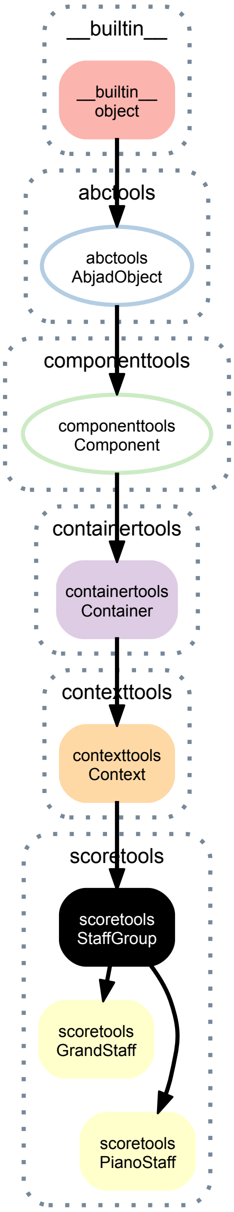
Score.**__radd__**(*expr*)
Extend container by contents of expr to the right.

Score.**__repr__**()

Score.**__rmul__**(*n*)

Score.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

28.1.7 scoretools.StaffGroup



class `scoretools.StaffGroup` (*music=None*, *context_name='StaffGroup'*, *name=None*)

Abjad model of staff group:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'1")
>>> staff_2 = Staff("g2 f2 e1")
```

```
>>> staff_group = scoretools.StaffGroup([staff_1, staff_2])
```

```
>>> f(staff_group)
\new StaffGroup <<
    \new Staff {
        c'4
        d'4
        e'4
        f'4
        g'1
    }
    \new Staff {
        g2
        f2
        e1
```

```
    }
>>
```

Return staff group.

Read-only properties

StaffGroup.**contents_duration**

StaffGroup.**descendants**
Read-only reference to component descendants score selection.

StaffGroup.**duration**

StaffGroup.**duration_in_seconds**

StaffGroup.**engraver_consists**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

StaffGroup.**engraver_removals**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

StaffGroup.**is_semantic**

StaffGroup.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

StaffGroup.**lilypond_format**

StaffGroup.**lineage**
Read-only reference to component lineage score selection.

StaffGroup.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

StaffGroup.**override**
Read-only reference to LilyPond grob override component plug-in.

StaffGroup.**parent**

StaffGroup.**parentage**
Read-only reference to component parentage score selection.

StaffGroup.**preprolated_duration**

StaffGroup.**prolation**

StaffGroup.**set**
Read-only reference LilyPond context setting component plug-in.

StaffGroup.**spanners**
Read-only reference to unordered set of spanners attached to component.

StaffGroup.**storage_format**
Storage format of Abjad object.

Return string.

StaffGroup.**timespan**
Read-only timespan of component.

StaffGroup.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

StaffGroup.**context_name**
Read / write name of context as a string.

StaffGroup.**is_nonsemantic**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

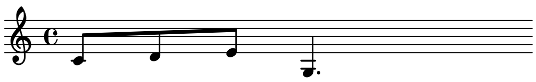
StaffGroup.**is_parallel**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```

Return none.

StaffGroup.**name**

Read-write name of context. Must be string or none.

Methods

StaffGroup.**append**(*component*)

Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

`StaffGroup.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

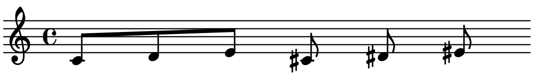
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`StaffGroup.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`StaffGroup.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

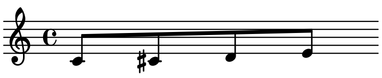
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`StaffGroup.`**`pop`**(*i=-1*)

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

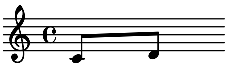
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

StaffGroup.**remove**(*component*)
Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

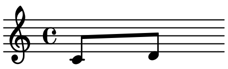
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

StaffGroup.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

StaffGroup.**__contains__**(*expr*)
True if expr is in container, otherwise False.

StaffGroup.**__copy__**(**args*)

StaffGroup.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

StaffGroup.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

StaffGroup.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StaffGroup.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

StaffGroup.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

StaffGroup.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

StaffGroup.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

StaffGroup.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StaffGroup.**__len__**()
Return nonnegative integer number of components in container.

StaffGroup.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

StaffGroup.**__mul__**(*n*)

StaffGroup.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

StaffGroup.**__radd__**(*expr*)
Extend container by contents of expr to the right.

StaffGroup.**__repr__**()

StaffGroup.**__rmul__**(*n*)

StaffGroup.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

28.2 Functions

28.2.1 scoretools.add_double_bar_to_end_of_score

scoretools.**add_double_bar_to_end_of_score**(*score*)
New in version 2.0. Add double bar to end of *score*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
```

```
>>> scoretools.add_double_bar_to_end_of_score(staff)
BarLine('|.')(f'4)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    \bar "|."
}
```

```
>>> show(staff)
```

Return double bar.

28.2.2 scoretools.add_markup_to_end_of_score

`scoretools.add_markup_to_end_of_score`(*score*, *markup*, *extra_offset=None*)

New in version 2.0. Add *markup* to end of *score*:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> markup = r'\italic \right-column { "Bremen - Boston - LA." "Jul 2010 - May 2011." }'
>>> markup = markuptools.Markup(markup, Down)
>>> markup = scoretools.add_markup_to_end_of_score(staff, markup, (4, -2))
```

```
>>> z(markup)
markuptools.Markup((
    markuptools.MarkupCommand(
        'italic',
        markuptools.MarkupCommand(
            'right-column',
            [
                'Bremen - Boston - LA.',
                'Jul 2010 - May 2011.'
            ]
            )
        ),
    ),
    direction=Down
    )
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    \once \override TextScript #'extra-offset = #'(4 . -2)
    f'4 _ \markup {
        \italic
            \right-column
                {
                    "Bremen - Boston - LA."
                    "Jul 2010 - May 2011."
                }
        }
}
```

```
>>> show(staff)
```

Return *markup*.

28.2.3 scoretools.all_are_scores

`scoretools.all_are_scores(expr)`

New in version 2.6. True when *expr* is a sequence of Abjad scores:

```
>>> score = Score([Staff([Note("c'4")])])
```

```
>>> score
Score<<1>>
```

```
>>> scoretools.all_are_scores([score])
True
```

True when *expr* is an empty sequence:

```
>>> scoretools.all_are_scores([])
True
```

Otherwise false:

```
>>> scoretools.all_are_scores('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

28.2.4 scoretools.get_first_score_in_improper_parentage_of_component

`scoretools.get_first_score_in_improper_parentage_of_component(component)`

New in version 2.0. Get first score in improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score = Score([staff])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> scoretools.get_first_score_in_improper_parentage_of_component(score.leaves[0])
Score<<1>>
```

Return score or none.

28.2.5 scoretools.get_first_score_in_proper_parentage_of_component

`scoretools.get_first_score_in_proper_parentage_of_component(component)`

New in version 2.0. Get first score in proper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> score = Score([staff])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
>>
```

```
>>> scoretools.get_first_score_in_proper_parentage_of_component(score.leaves[0])
Score<<1>>
```

Return score or none.

28.2.6 scoretools.list_performer_names

`scoretools.list_performer_names(locale='en-us')`

New in version 2.5. List performer names:

```
>>> for performer_name in scoretools.list_performer_names():
...     performer_name
...
'accordionist'
'baritone'
'bass'
'bassist'
'bassoonist'
'cellist'
'clarinetist'
'contralto'
'flutist'
'guitarist'
'harpist'
'harpsichordist'
'hornist'
'mezzo-soprano'
'oboist'
'percussionist'
'pianist'
'saxophonist'
'soprano'
'tenor'
'trombonist'
'trumpeter'
'tubist'
'vibraphonist'
'violinist'
'violist'
'xylophonist'
```

Available values for *locale* are `'en-us'` and `'en-uk'`.

28.2.7 scoretools.list_primary_performer_names

`scoretools.list_primary_performer_names()`

New in version 2.5. List performer names:

```
>>> for pair in scoretools.list_primary_performer_names():
...     pair
...
('accordionist', 'acc.')
('baritone', 'baritone')
('bass', 'bass')
('bassist', 'vb.')
('bassoonist', 'bsn.')
('cellist', 'vc.')
```

```
('clarinetist', 'cl. in B-flat')
('contralto', 'contralto')
('flutist', 'fl.')
('guitarist', 'gt.')
('harpist', 'hp.')
('harpsichordist', 'hpschd.')
('hornist', 'hn.')
('mezzo-soprano', 'mezzo-soprano')
('oboist', 'ob.')
('pianist', 'pf.')
('saxophonist', 'alto sax.')
('soprano', 'soprano')
('tenor', 'tenor')
('trombonist', 'ten. trb.')
('trumpeter', 'tp.')
('tubist', 'tb.')
('violinist', 'vn.')
('violist', 'va.')
```

Return list.

28.2.8 scoretools.make_empty_piano_score

scoretools.**make_empty_piano_score**()

New in version 1.1. Make empty piano score:

```
>>> score, treble, bass = scoretools.make_empty_piano_score()
```

```
>>> f(score)
\new Score <<
    \new PianoStaff <<
        \context Staff = "treble" {
            \clef "treble"
        }
        \context Staff = "bass" {
            \clef "bass"
        }
    >>
>>
```

Return score, treble staff, bass staff.

28.2.9 scoretools.make_piano_score_from_leaves

scoretools.**make_piano_score_from_leaves**(*leaves*, *lowest_treble_pitch=None*)

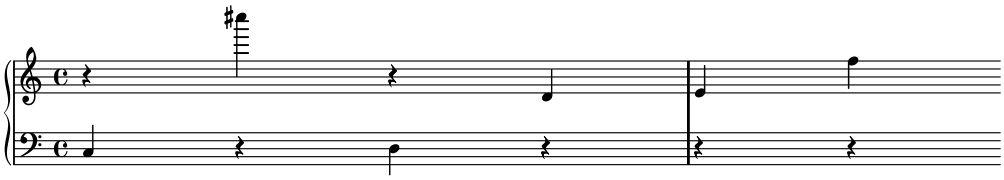
New in version 2.0. Make piano score from *leaves*:

```
>>> notes = [Note(x, (1, 4)) for x in [-12, 37, -10, 2, 4, 17]]
>>> score, treble_staff, bass_staff = scoretools.make_piano_score_from_leaves(notes)
```

```
>>> f(score)
\new Score <<
    \new PianoStaff <<
        \context Staff = "treble" {
            \clef "treble"
            r4
            cs''''4
            r4
            d'4
            e'4
            f''4
        }
        \context Staff = "bass" {
            \clef "bass"
            c4
            r4
            d4
```

```
                r4
                r4
                r4
            }
        >>
    >>
```

```
>>> show(score)
```



When `lowest_treble_pitch=None` set to B3.

Return score, treble staff, bass staff.

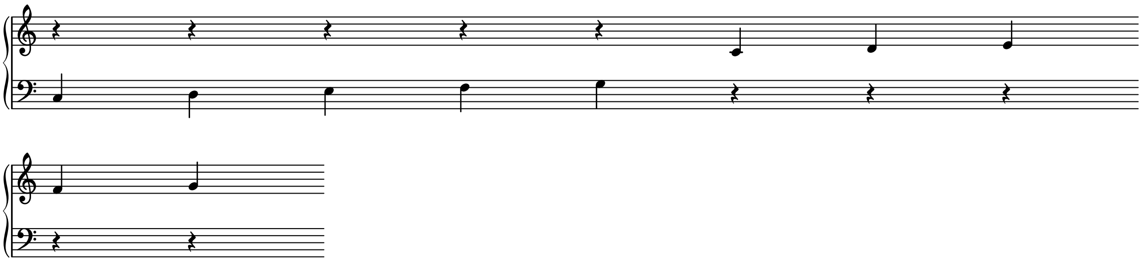
28.2.10 scoretools.make_piano_sketch_score_from_leaves

scoretools.**make_piano_sketch_score_from_leaves**(*leaves*, *lowest_treble_pitch=None*)
New in version 2.0. Make piano sketch score from *leaves*:

```
>>> notes = notetools.make_notes([-12, -10, -8, -7, -5, 0, 2, 4, 5, 7], [(1, 4)])
>>> score, treble_staff, bass_staff = scoretools.make_piano_sketch_score_from_leaves(notes)
```

```
>>> f(score)
\new Score \with {
    \override BarLine #'stencil = ##f
    \override BarNumber #'transparent = ##t
    \override SpanBar #'stencil = ##f
    \override TimeSignature #'stencil = ##f
} <<
    \new PianoStaff <<
        \context Staff = "treble" {
            \clef "treble"
            #(set-accidental-style 'forget)
            r4
            r4
            r4
            r4
            r4
            c'4
            d'4
            e'4
            f'4
            g'4
        }
        \context Staff = "bass" {
            \clef "bass"
            #(set-accidental-style 'forget)
            c4
            d4
            e4
            f4
            g4
            r4
            r4
            r4
            r4
            r4
        }
    >>
>>
```

```
>>> show(score)
```



When `lowest_treble_pitch=None` set to B3.

Make time signatures and bar numbers transparent.

Do not print bar lines or span bars.

Set all staff accidental styles to forget.

Return score, treble staff, bass staff.

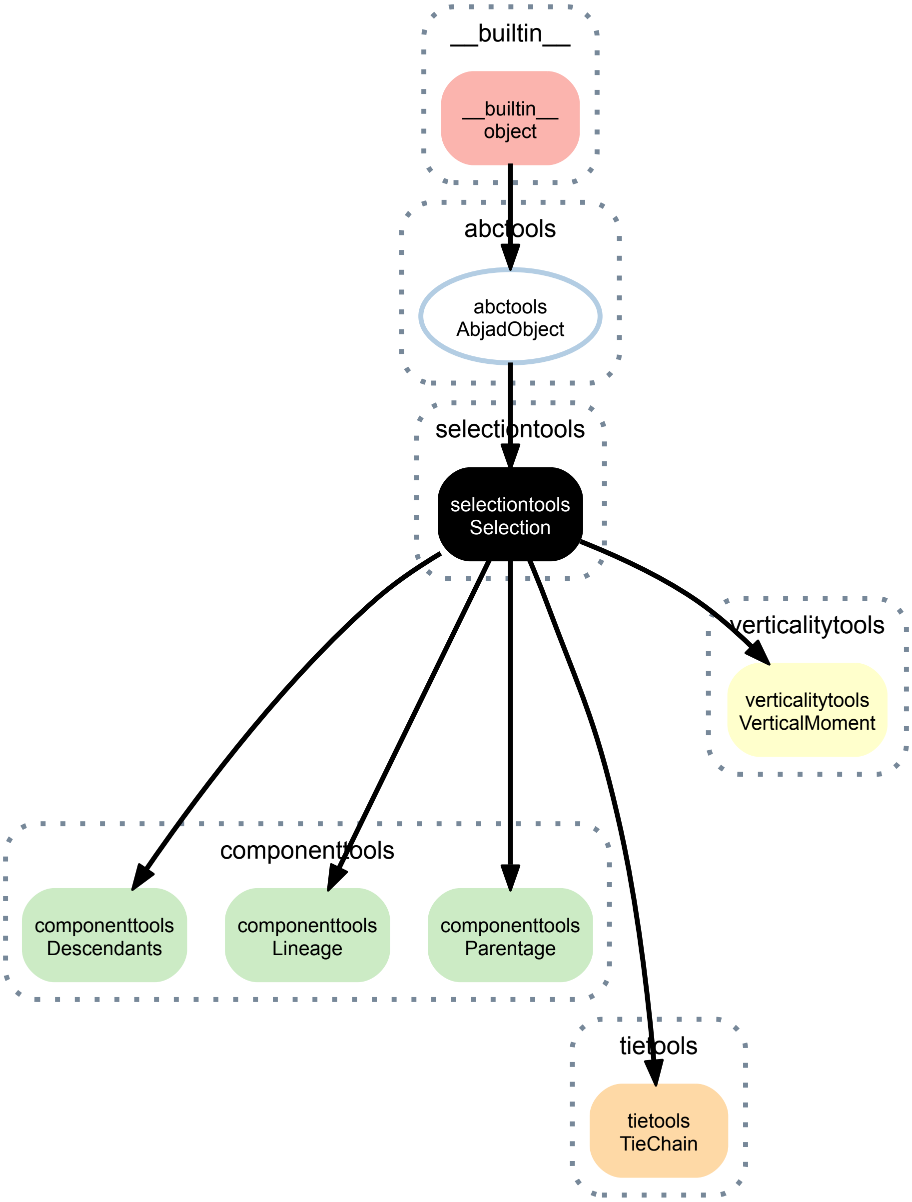
CHAPTER

TWENTYNINE

SELECTIONTOOLS

29.1 Concrete Classes

29.1.1 selectiontools.Selection



class `selectiontools.Selection(music)`

New in version 2.9. Selection taken from a single score.

Selection objects will eventually pervade the system and model all user selections.

This means that selection objects will eventually serve as input to most functions in the API. Selection objects will also eventually be returned as output from most functions in the API.

Selections are immutable and never change after instantiation.

Read-only properties

`Selection.music`
Read-only tuple of components in selection.

`Selection.storage_format`
Storage format of Abjad object.

Return string.

`Selection.timespan`
Read-only timespan of selection.

Special methods

`Selection.__add__(expr)`

`Selection.__contains__(expr)`

`Selection.__eq__(expr)`

`Selection.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Selection.__getitem__(expr)`

`Selection.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`Selection.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Selection.__len__()`

`Selection.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`Selection.__ne__(expr)`

`Selection.__radd__(expr)`

`Selection.__repr__()`

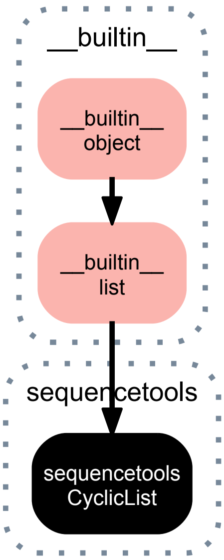
CHAPTER

THIRTY

SEQUENCETOOLS

30.1 Concrete Classes

30.1.1 sequencetools.CyclicList



class sequencetools.**CyclicList**

New in version 2.0. Abjad model of cyclic list:

```
>>> cyclic_list = sequencetools.CyclicList('abcd')
```

```
>>> cyclic_list
CyclicList([a, b, c, d])
```

```
>>> for x in range(8):
...     print x, cyclic_list[x]
...
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
```

Cyclic lists overload the item-getting method of built-in lists.

Cyclic lists return a value for any integer index.

Cyclic lists otherwise behave exactly like built-in lists.

Methods

`CyclicList.append()`
L.append(object) – append object to end

`CyclicList.count(value)` → integer – return number of occurrences of value

`CyclicList.extend()`
L.extend(iterable) – extend list by appending elements from the iterable

`CyclicList.index(value[, start[, stop]])` → integer – return first index of value.
Raises ValueError if the value is not present.

`CyclicList.insert()`
L.insert(index, object) – insert object before index

`CyclicList.pop([index])` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`CyclicList.remove()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`CyclicList.reverse()`
L.reverse() – reverse *IN PLACE*

`CyclicList.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`CyclicList.__add__()`
x.__add__(y) <==> x+y

`CyclicList.__contains__()`
x.__contains__(y) <==> y in x

`CyclicList.__delitem__()`
x.__delitem__(y) <==> del x[y]

`CyclicList.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`CyclicList.__eq__()`
x.__eq__(y) <==> x==y

`CyclicList.__ge__()`
x.__ge__(y) <==> x>=y

`CyclicList.__getitem__(expr)`

`CyclicList.__getslice__(start_index, stop_index)`

`CyclicList.__gt__()`
x.__gt__(y) <==> x>y

`CyclicList.__iadd__()`
x.__iadd__(y) <==> x+=y

`CyclicList.__imul__()`
x.__imul__(y) <==> x*=y

`CyclicList.__iter__()` <==> *iter(x)*

`CyclicList.__le__()`
x.__le__(y) <==> x<=y

CyclicList.**__len__**() <==> *len(x)*

CyclicList.**__lt__**()
x.__lt__(y) <==> x<y

CyclicList.**__mul__**()
x.__mul__(n) <==> x*n

CyclicList.**__ne__**()
x.__ne__(y) <==> x!=y

CyclicList.**__repr__**()

CyclicList.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

CyclicList.**__rmul__**()
x.__rmul__(n) <==> n*x

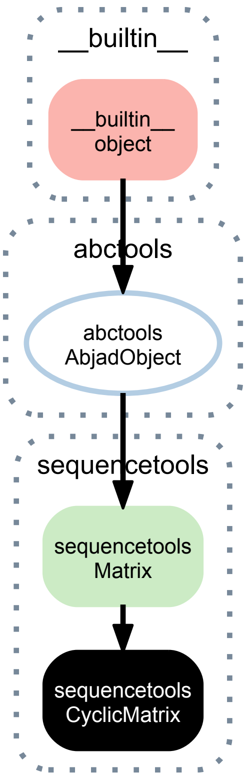
CyclicList.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

CyclicList.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

CyclicList.**__str__**()

30.1.2 sequencetools.CyclicMatrix



class sequencetools.**CyclicMatrix**(**args*, ***kwargs*)
New in version 2.0. Abjad model of cyclic matrix.

Initialize from rows:

```
>>> cyclic_matrix = sequencetools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix
CyclicMatrix(3x4)
```

```
>>> cyclic_matrix[2]
CyclicTuple([20, 21, 22, 23])
```

```
>>> cyclic_matrix[2][2]
22
```

```
>>> cyclic_matrix[99]
CyclicTuple([0, 1, 2, 3])
```

```
>>> cyclic_matrix[99][99]
3
```

Initialize from columns:

```
>>> cyclic_matrix = sequencetools.CyclicMatrix(
...     columns=[[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]])
```

```
>>> cyclic_matrix
CyclicMatrix(3x4)
```

```
>>> cyclic_matrix[2]
CyclicTuple([20, 21, 22, 23])
```

```
>>> cyclic_matrix[2][2]
22
```

```
>>> cyclic_matrix[99]
CyclicTuple([0, 1, 2, 3])
```

```
>>> cyclic_matrix[99][99]
3
```

CyclicMatrix implements only item retrieval in this revision.

Concatenation and division remain to be implemented.

Standard transforms of linear algebra remain to be implemented.

Read-only properties

`CyclicMatrix.`**`columns`**

Read-only columns:

```
>>> cyclic_matrix = sequencetools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix.columns
CyclicTuple([[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]])
```

Return cyclic tuple.

`CyclicMatrix.`**`rows`**

Read-only rows:

```
>>> cyclic_matrix = sequencetools.CyclicMatrix(
...     [[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> cyclic_matrix.rows
CyclicTuple([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

Return cyclic tuple.

`CyclicMatrix.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`CyclicMatrix.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`CyclicMatrix.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`CyclicMatrix.`**`__getitem__`**`(`*expr*`)`

`CyclicMatrix.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`CyclicMatrix.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`CyclicMatrix.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

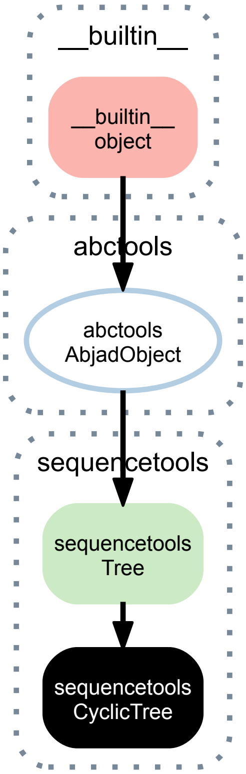
Raise exception.

`CyclicMatrix.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`CyclicMatrix.`**`__repr__`**`()`

30.1.3 sequencetools.CyclicTree



class sequencetools.**CyclicTree**(*expr*)

New in version 2.5. Abjad data structure to work with a sequence whose elements have been grouped into arbitrarily many levels of cyclic containment.

Exactly like the `Tree` class but with the additional affordance that all integer indices of any size work at every level of structure.

Cyclic trees raise no index errors.

Here is a cyclic tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = sequencetools.CyclicTree(sequence)
```

```
>>> cyclic_tree
CyclicTree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Here's an internal node:

```
>>> cyclic_tree[2]
CyclicTree([4, 5])
```

Here's the same node indexed with a different way:

```
>>> cyclic_tree[2]
CyclicTree([4, 5])
```

With a negative index:

```
>>> cyclic_tree[-2]
CyclicTree([4, 5])
```

And another negative index:

```
>>> cyclic_tree[-6]
CyclicTree([4, 5])
```

Here's a leaf node:

```
>>> cyclic_tree[2][0]
CyclicTree(4)
```

And here's the same node indexed a different way:

```
>>> cyclic_tree[2][20]
CyclicTree(4)
```

All other interface attributes function as in `Tree`.

Read-only properties

`CyclicTree.`**children**

Children of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].children
(Tree(2), Tree(3))
```

Return tuple of zero or more nodes.

`CyclicTree.`**depth**

Depth of subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].depth
2
```

Return nonnegative integer.

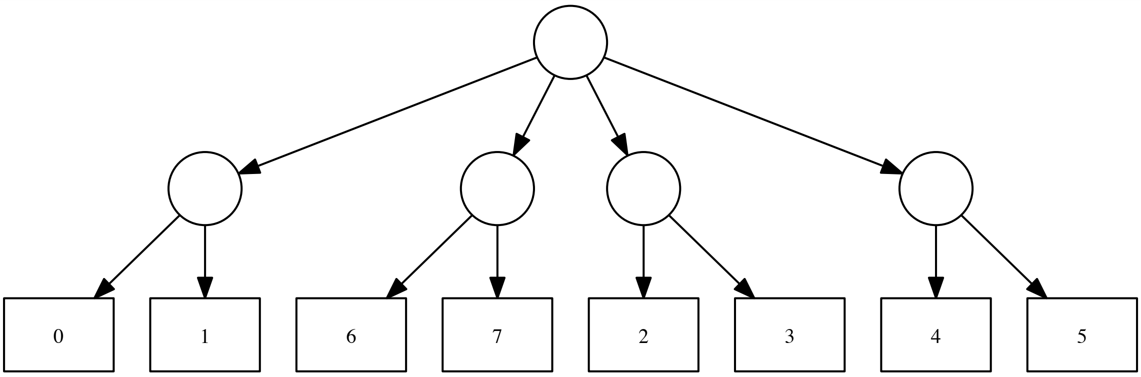
`CyclicTree.`**graphviz_format**

`CyclicTree.`**graphviz_graph**

The GraphvizGraph representation of the Tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> graph = tree.graphviz_graph
>>> iotools.graph(graph)
```



Return graphviz graph.

`CyclicTree.`**improper_parentage**

Improper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].improper_parentage
(Tree([2, 3]), Tree([[0, 1], [2, 3], [4, 5], [6, 7]]))
```

Return tuple of one or more nodes.

CyclicTree.**index_in_parent**
Index of node in parent of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].index_in_parent
1
```

Return nonnegative integer.

CyclicTree.**level**
Level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].level
1
```

Return nonnegative integer.

CyclicTree.**manifest_payload**
Manifest payload of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree.manifest_payload
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> tree[-1].manifest_payload
[6, 7]
```

```
>>> tree[-1][-1].manifest_payload
[7]
```

Return list.

CyclicTree.**negative_level**
Negative level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].negative_level
-2
```

Return negative integer.

CyclicTree.**payload**
Payload of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Return none for interior node:

```
>>> tree.payload is None
True
```

```
>>> tree[-1].payload is None
True
```

Return unwrapped payload for leaf node:

```
>>> tree[-1][-1].payload
7
```

Return arbitrary expression or none.

CyclicTree.**position**
Position of node relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].position
(1,)
```

Return tuple of zero or more nonnegative integers.

CyclicTree.**proper_parentage**
Proper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].proper_parentage
(Tree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Return tuple of zero or more nodes.

CyclicTree.**root**
Root of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].proper_parentage
(Tree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Return node.

CyclicTree.**storage_format**
Tree storage format:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> z(tree)
sequencetools.Tree(
    [[0, 1], [2, 3], [4, 5], [6, 7]]
    )
```

Return string.

CyclicTree.**width**
Number of leaves in subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].width
2
```

Return nonnegative integer.

Methods

`CyclicTree.`**`get_manifest_payload_of_next_n_nodes_at_level`**(*n*, *level*)

Get manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Get manifest paylaod of next 4 nodes at level 2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, 2)
[1, 2, 3, 4]
```

Get manifest paylaod of next 3 nodes at level 1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, 1)
[1, 2, 3, 4, 5]
```

Get manifest paylaod of next node at level 0:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(1, 0)
[1, 2, 3, 4, 5, 6, 7]
```

Get manifest paylaod of next 4 nodes at level -1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, -1)
[1, 2, 3, 4]
```

Get manifest paylaod of next 3 nodes at level -2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, -2)
[1, 2, 3, 4, 5]
```

Get manifest paylaod of previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, 2)
[6, 5, 4, 3]
```

Get manifest paylaod of previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, 1)
[6, 5, 4, 3, 2]
```

Get manifest paylaod of previous node at level 0:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-1, 0)
[6, 5, 4, 3, 2, 1, 0]
```

Get manifest paylaod of previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, -1)
[6, 5, 4, 3]
```

Get manifest paylaod of previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, -2)
[6, 5, 4, 3, 2]
```

Trim first node if necessary.

Return list of arbitrary values.

`CyclicTree.`**`get_next_n_complete_nodes_at_level`**(*n*, *level*)

Get next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, 2)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, 1)
[Tree([1]), Tree([2, 3]), Tree([4, 5]), Tree([6, 7])]
```

Get next 4 nodes at level -1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, -1)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, -2)
[Tree([1]), Tree([2, 3]), Tree([4, 5]), Tree([6, 7])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, 2)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, 1)
[Tree([6]), Tree([4, 5]), Tree([2, 3]), Tree([0, 1])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, -1)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, -2)
[Tree([6]), Tree([4, 5]), Tree([2, 3]), Tree([0, 1])]
```

Trim first node if necessary.

Return list of nodes.

`CyclicTree.`**`get_next_n_nodes_at_level`**(*n*, *level*)

Get next *n* nodes at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.CyclicTree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, 2)
[CyclicTree(1), CyclicTree(2), CyclicTree(3), CyclicTree(4)]
```

Get next 10 nodes at level 2:

```
>>> for node in tree[0][0].get_next_n_nodes_at_level(10, 2):
...     node
CyclicTree(1)
CyclicTree(2)
CyclicTree(3)
CyclicTree(4)
CyclicTree(5)
CyclicTree(6)
CyclicTree(7)
CyclicTree(1)
CyclicTree(2)
CyclicTree(3)
```

PREVIOUS MNODES

Get previous 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(-4, 2)
[CyclicTree(7), CyclicTree(6), CyclicTree(5), CyclicTree(4)]
```

Get previous 10 nodes at level 2:

```
>>> for node in tree[0][0].get_next_n_nodes_at_level(-10, 2):
...     node
CyclicTree(7)
CyclicTree(6)
CyclicTree(5)
CyclicTree(4)
CyclicTree(3)
CyclicTree(2)
CyclicTree(1)
CyclicTree(7)
CyclicTree(6)
CyclicTree(5)
```

Return list of nodes.

CyclicTree.**get_node_at_position**(*position*)
Get node at *position*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = sequencetools.CyclicTree(sequence)
```

```
>>> cyclic_tree.get_node_at_position((2, 1))
CyclicTree(5)
```

```
>>> cyclic_tree.get_node_at_position((2, 99))
CyclicTree(5)
```

```
>>> cyclic_tree.get_node_at_position((82, 1))
CyclicTree(5)
```

```
>>> cyclic_tree.get_node_at_position((82, 99))
CyclicTree(5)
```

Return node.

CyclicTree.**get_position_of_descendant**(*descendant*)
Get position of *descendent* relative to node rather than relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[3].get_position_of_descendant(tree[3][0])
(0,)
```

Return tuple of zero or more nonnegative integers.

CyclicTree.**index**(*node*)
Index of *node*:

```
>>> sequence = [0, 1, 2, 2, 3, 4]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for node in tree:
...     node, tree.index(node)
(Tree(0), 0)
(Tree(1), 1)
(Tree(2), 2)
(Tree(2), 3)
(Tree(3), 4)
(Tree(4), 5)
```

Return nonnegative integer.

CyclicTree.**is_at_level**(*level*)
True when node is at *level* in containing tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1][1].is_at_level(-1)
True
```

False otherwise:

```
>>> tree[1][1].is_at_level(0)
False
```

Works for positive, negative and zero-valued *level*.

Return boolean.

`CyclicTree.`**`iterate_at_level`**(*level*, *reverse=False*)

Iterate tree at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Left-to-right examples:

```
>>> for x in tree.iterate_at_level(0): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1): x
...
Tree([0, 1])
Tree([2, 3])
Tree([4, 5])
Tree([6, 7])
```

```
>>> for x in tree.iterate_at_level(2): x
...
Tree(0)
Tree(1)
Tree(2)
Tree(3)
Tree(4)
Tree(5)
Tree(6)
Tree(7)
```

```
>>> for x in tree.iterate_at_level(-1): x
...
Tree(0)
Tree(1)
Tree(2)
Tree(3)
Tree(4)
Tree(5)
Tree(6)
Tree(7)
```

```
>>> for x in tree.iterate_at_level(-2): x
...
Tree([0, 1])
Tree([2, 3])
Tree([4, 5])
Tree([6, 7])
```

```
>>> for x in tree.iterate_at_level(-3): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Right-to-left examples:

```
>>> for x in tree.iterate_at_level(0, reverse=True): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1, reverse=True): x
...
Tree([6, 7])
Tree([4, 5])
Tree([2, 3])
Tree([0, 1])
```

```
>>> for x in tree.iterate_at_level(2, reverse=True): x
...
Tree(7)
Tree(6)
Tree(5)
Tree(4)
Tree(3)
Tree(2)
Tree(1)
Tree(0)
```

```
>>> for x in tree.iterate_at_level(-1, reverse=True): x
...
Tree(7)
Tree(6)
Tree(5)
Tree(4)
Tree(3)
Tree(2)
Tree(1)
Tree(0)
```

```
>>> for x in tree.iterate_at_level(-2, reverse=True): x
...
Tree([6, 7])
Tree([4, 5])
Tree([2, 3])
Tree([0, 1])
```

```
>>> for x in tree.iterate_at_level(-3, reverse=True): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Return node generator.

`CyclicTree.`**`iterate_depth_first`**(*reverse=False*)

Iterate tree depth-first:

Example 1. Iterate tree depth-first from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for node in tree.iterate_depth_first(): node
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
Tree([0, 1])
Tree(0)
Tree(1)
Tree([2, 3])
Tree(2)
Tree(3)
Tree([4, 5])
Tree(4)
Tree(5)
Tree([6, 7])
Tree(6)
Tree(7)
```

Example 2. Iterate tree depth-first from right to left:

```
>>> for node in tree.iterate_depth_first(reverse=True): node
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
Tree([6, 7])
Tree(7)
Tree(6)
Tree([4, 5])
Tree(5)
Tree(4)
Tree([2, 3])
Tree(3)
Tree(2)
Tree([0, 1])
Tree(1)
Tree(0)
```

Return node generator.

CyclicTree.**iterate_forever_depth_first**(*reverse=False*)

Iterate tree depth first.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> cyclic_tree = sequencetools.CyclicTree(sequence)
```

Example 1. Iterate from left to right:

```
>>> generator = cyclic_tree.iterate_forever_depth_first()
>>> for i in range(20):
...     generator.next()
CyclicTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicTree([0, 1])
CyclicTree(0)
CyclicTree(1)
CyclicTree([2, 3])
CyclicTree(2)
CyclicTree(3)
CyclicTree([4, 5])
CyclicTree(4)
CyclicTree(5)
CyclicTree([6, 7])
CyclicTree(6)
CyclicTree(7)
CyclicTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicTree([0, 1])
CyclicTree(0)
CyclicTree(1)
CyclicTree([2, 3])
CyclicTree(2)
CyclicTree(3)
```

Example 2. Iterate from right to left:

```
>>> generator = cyclic_tree.iterate_forever_depth_first(reverse=True)
>>> for i in range(20):
...     generator.next()
CyclicTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicTree([6, 7])
CyclicTree(7)
CyclicTree(6)
CyclicTree([4, 5])
CyclicTree(5)
CyclicTree(4)
CyclicTree([2, 3])
CyclicTree(3)
CyclicTree(2)
CyclicTree([0, 1])
CyclicTree(1)
CyclicTree(0)
CyclicTree([[0, 1], [2, 3], [4, 5], [6, 7]])
CyclicTree([6, 7])
CyclicTree(7)
CyclicTree(6)
```

```
CyclicTree([4, 5])
CyclicTree(5)
CyclicTree(4)
```

Return node generator.

`CyclicTree.`**`iterate_payload`**`(`*reverse=False*`)`

Iterate tree payload:

Example 1. Iterate payload from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for element in tree.iterate_payload():
...     element
...
0
1
2
3
4
5
6
7
```

Example 2. Iterate payload from right to left:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for element in tree.iterate_payload(reverse=True):
...     element
...
7
6
5
4
3
2
1
0
```

Return payload generator.

`CyclicTree.`**`remove_node`**`(`*node*`)`

Remove *node* from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree.remove_node(tree[1])
```

```
>>> tree
Tree([[0, 1], [4, 5], [6, 7]])
```

Return none.

`CyclicTree.`**`remove_to_root`**`(`*reverse=False*`)`

Remove node and all nodes left of node to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[0][0].remove_to_root()
>>> tree
Tree([[1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[0][1].remove_to_root()
>>> tree
Tree([[2, 3], [4, 5], [6, 7]])
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[1].remove_to_root()
>>> tree
Tree([[4, 5], [6, 7]])
```

Modify in-place to root.

Return none.

`CyclicTree.to_nested_lists()`
Change tree to nested lists:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.to_nested_lists()
[[0, 1], [2, 3], [4, 5], [6, 7]]
```

Return list of lists.

Special methods

`CyclicTree.__contains__(expr)`
True when tree contains *expr*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[-1] in tree
True
```

Otherwise false:

```
>>> tree[-1][-1] in tree
False
```

Return boolean.

`CyclicTree.__eq__(expr)`
True when *expr* is the same type as tree and when the payload of all subtrees are equal:

```
>>> sequence_1 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_1 = sequencetools.Tree(sequence_1)
>>> sequence_2 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_2 = sequencetools.Tree(sequence_2)
>>> sequence_3 = [[0, 1], [2, 3], [4, 5]]
>>> tree_3 = sequencetools.Tree(sequence_3)
```

```
>>> tree_1 == tree_1
True
>>> tree_1 == tree_2
True
>>> tree_1 == tree_3
False
>>> tree_2 == tree_1
True
>>> tree_2 == tree_2
True
>>> tree_2 == tree_3
False
```

```
>>> tree_3 == tree_1
False
>>> tree_3 == tree_2
False
>>> tree_3 == tree_3
True
```

Return boolean.

CyclicTree.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CyclicTree.**__getitem__**(*expr*)
Get item from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[-1]
Tree([6, 7])
```

Get slice from tree:

```
>>> tree[-2:]
[Tree([4, 5]), Tree([6, 7])]
```

Return node.

CyclicTree.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

CyclicTree.**__iter__**()

CyclicTree.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CyclicTree.**__len__**()
Return the number of children in tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> len(tree)
4
```

Return nonnegative integer.

CyclicTree.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CyclicTree.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

CyclicTree.**__repr__**()
Interpreter representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Return string.

CyclicTree.**__str__**()

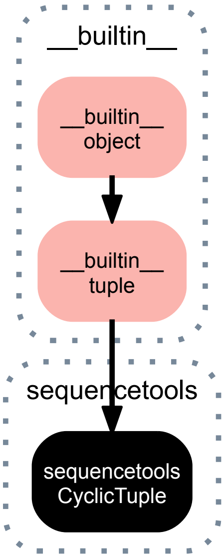
String representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> str(tree)
'[[0, 1], [2, 3], [4, 5], [6, 7]]'
```

Return string.

30.1.4 sequencetools.CyclicTuple



class sequencetools.**CyclicTuple**

New in version 2.0. Abjad model of cyclic tuple:

```
>>> cyclic_tuple = sequencetools.CyclicTuple('abcd')
```

```
>>> cyclic_tuple
CyclicTuple([a, b, c, d])
```

```
>>> for x in range(8):
...     print x, cyclic_tuple[x]
...
0 a
1 b
2 c
3 d
4 a
5 b
6 c
7 d
```

Cyclic tuples overload the item-getting method of built-in tuples.

Cyclic tuples return a value for any integer index.

Cyclic tuples otherwise behave exactly like built-in tuples.

Methods

`CyclicTuple.`**`count`**(*value*) → integer – return number of occurrences of value

`CyclicTuple.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Special methods

`CyclicTuple.`**`__add__`**()
x.__add__(y) <==> x+y

`CyclicTuple.`**`__contains__`**()
x.__contains__(y) <==> y in x

`CyclicTuple.`**`__eq__`**()
x.__eq__(y) <==> x==y

`CyclicTuple.`**`__ge__`**()
x.__ge__(y) <==> x>=y

`CyclicTuple.`**`__getitem__`**(*expr*)

`CyclicTuple.`**`__getslice__`**(*start_index*, *stop_index*)

`CyclicTuple.`**`__gt__`**()
x.__gt__(y) <==> x>y

`CyclicTuple.`**`__hash__`**() <==> *hash(x)*

`CyclicTuple.`**`__iter__`**() <==> *iter(x)*

`CyclicTuple.`**`__le__`**()
x.__le__(y) <==> x<=y

`CyclicTuple.`**`__len__`**() <==> *len(x)*

`CyclicTuple.`**`__lt__`**()
x.__lt__(y) <==> x<y

`CyclicTuple.`**`__mul__`**()
x.__mul__(n) <==> x*n

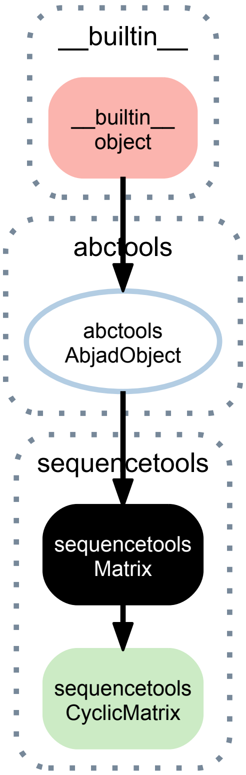
`CyclicTuple.`**`__ne__`**()
x.__ne__(y) <==> x!=y

`CyclicTuple.`**`__repr__`**()

`CyclicTuple.`**`__rmul__`**()
x.__rmul__(n) <==> n*x

`CyclicTuple.`**`__str__`**()

30.1.5 sequencetools.Matrix



class `sequencetools.Matrix`(**args*, ***kwargs*)

New in version 2.0. Abjad model of matrix.

Initialize from rows:

```
>>> matrix = sequencetools.Matrix([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix
Matrix(3x4)
```

```
>>> matrix[:]
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

```
>>> matrix[2]
(20, 21, 22, 23)
```

```
>>> matrix[2][0]
20
```

Initialize from columns:

```
>>> matrix = sequencetools.Matrix(
...     columns=[[0, 10, 20], [1, 11, 21], [2, 12, 22], [3, 13, 23]])
```

```
>>> matrix
Matrix(3x4)
```

```
>>> matrix[:]
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

```
>>> matrix[2]
(20, 21, 22, 23)
```

```
>>> matrix[2][0]
20
```

Matrix implements only item retrieval in this revision.

Concatenation and division remain to be implemented.

Standard transforms of linear algebra remain to be implemented.

Read-only properties

Matrix.**columns**
Read-only columns:

```
>>> matrix = sequencetools.Matrix([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix.columns
((0, 10, 20), (1, 11, 21), (2, 12, 22), (3, 13, 23))
```

Return tuple.

Matrix.**rows**
Read-only rows:

```
>>> matrix = sequencetools.Matrix([[0, 1, 2, 3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

```
>>> matrix.rows
((0, 1, 2, 3), (10, 11, 12, 13), (20, 21, 22, 23))
```

Return tuple.

Matrix.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

Matrix.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Matrix.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Matrix.**__getitem__**(*expr*)

Matrix.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Matrix.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Matrix.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

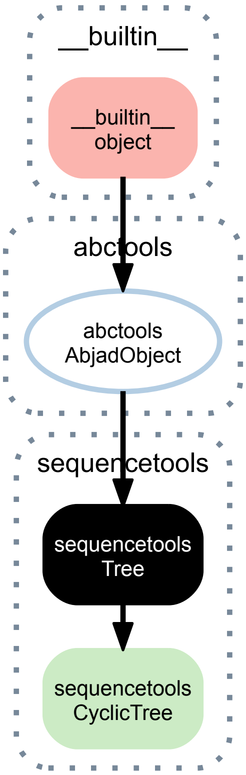
Raise exception.

Matrix.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Matrix.**__repr__**()

30.1.6 sequencetools.Tree



class sequencetools.**Tree** (*expr*)

New in version 2.4. Abjad data structure to work with a sequence whose elements have been grouped into arbitrarily many levels of containment.

Here is a tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.parent is None
True
```

```
>>> tree.children
(Tree([0, 1]), Tree([2, 3]), Tree([4, 5]), Tree([6, 7]))
```

```
>>> tree.depth
3
```

Here's an internal node:

```
>>> tree[2]
Tree([4, 5])
```

```
>>> tree[2].parent
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree[2].children
(Tree(4), Tree(5))
```

```
>>> tree[2].depth
2
```

```
>>> tree[2].level
1
```

Here's a leaf node:

```
>>> tree[2][0]
Tree(4)
```

```
>>> tree[2][0].parent
Tree([4, 5])
```

```
>>> tree[2][0].children
()
```

```
>>> tree[2][0].depth
1
```

```
>>> tree[2][0].level
2
```

```
>>> tree[2][0].position
(2, 0)
```

```
>>> tree[2][0].payload
4
```

Only leaf nodes carry payload. Internal nodes carry no payload.

Negative levels are available to work with trees bottom-up instead of top-down.

Trees do not yet implement append or extend methods.

Read-only properties

`Tree.`**`children`**

Children of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].children
(Tree(2), Tree(3))
```

Return tuple of zero or more nodes.

`Tree.`**`depth`**

Depth of subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].depth
2
```

Return nonnegative integer.

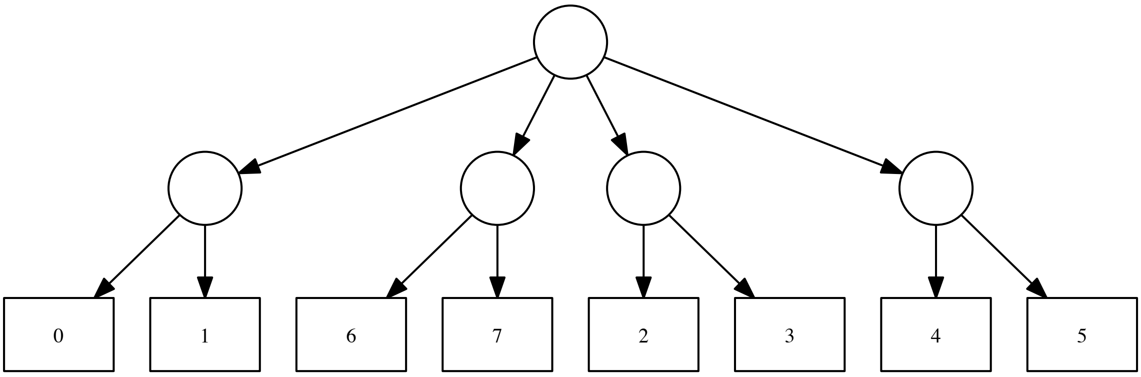
`Tree.`**`graphviz_format`**

`Tree.`**`graphviz_graph`**

The GraphvizGraph representation of the Tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> graph = tree.graphviz_graph
>>> iotools.graph(graph)
```



Return graphviz graph.

Tree.**improper_parentage**
Improper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].improper_parentage
(Tree([2, 3]), Tree([[0, 1], [2, 3], [4, 5], [6, 7]]))
```

Return tuple of one or more nodes.

Tree.**index_in_parent**
Index of node in parent of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].index_in_parent
1
```

Return nonnegative integer.

Tree.**level**
Level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].level
1
```

Return nonnegative integer.

Tree.**manifest_payload**
Manifest payload of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree.manifest_payload
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> tree[-1].manifest_payload
[6, 7]
```

```
>>> tree[-1][-1].manifest_payload
[7]
```

Return list.

Tree.**negative_level**
Negative level of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].negative_level
-2
```

Return negative integer.

`Tree.payload`
Payload of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Return none for interior node:

```
>>> tree.payload is None
True
```

```
>>> tree[-1].payload is None
True
```

Return unwrapped payload for leaf node:

```
>>> tree[-1][-1].payload
7
```

Return arbitrary expression or none.

`Tree.position`
Position of node relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].position
(1,)
```

Return tuple of zero or more nonnegative integers.

`Tree.proper_parentage`
Proper parentage of node:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].proper_parentage
(Tree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Return tuple of zero or more nodes.

`Tree.root`
Root of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].proper_parentage
(Tree([[0, 1], [2, 3], [4, 5], [6, 7]]),)
```

Return node.

`Tree.storage_format`
Tree storage format:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> z(tree)
sequencetools.Tree(
    [[0, 1], [2, 3], [4, 5], [6, 7]]
    )
```

Return string.

`Tree.width`
Number of leaves in subtree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1].width
2
```

Return nonnegative integer.

Methods

`Tree.get_manifest_payload_of_next_n_nodes_at_level(n, level)`
Get manifest payload of next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Get manifest paylaod of next 4 nodes at level 2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, 2)
[1, 2, 3, 4]
```

Get manifest paylaod of next 3 nodes at level 1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, 1)
[1, 2, 3, 4, 5]
```

Get manifest paylaod of next node at level 0:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(1, 0)
[1, 2, 3, 4, 5, 6, 7]
```

Get manifest paylaod of next 4 nodes at level -1:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(4, -1)
[1, 2, 3, 4]
```

Get manifest paylaod of next 3 nodes at level -2:

```
>>> tree[0][0].get_manifest_payload_of_next_n_nodes_at_level(3, -2)
[1, 2, 3, 4, 5]
```

Get manifest paylaod of previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, 2)
[6, 5, 4, 3]
```

Get manifest paylaod of previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, 1)
[6, 5, 4, 3, 2]
```

Get manifest paylaod of previous node at level 0:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-1, 0)
[6, 5, 4, 3, 2, 1, 0]
```

Get manifest paylaod of previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-4, -1)
[6, 5, 4, 3]
```

Get manifest paylaod of previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_manifest_payload_of_next_n_nodes_at_level(-3, -2)
[6, 5, 4, 3, 2]
```

Trim first node if necessary.

Return list of arbitrary values.

Tree.**get_next_n_complete_nodes_at_level**(*n*, *level*)

Get next *n* complete nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, 2)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, 1)
[Tree([1]), Tree([2, 3]), Tree([4, 5]), Tree([6, 7])]
```

Get next 4 nodes at level -1:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(4, -1)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_complete_nodes_at_level(3, -2)
[Tree([1]), Tree([2, 3]), Tree([4, 5]), Tree([6, 7])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, 2)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, 1)
[Tree([6]), Tree([4, 5]), Tree([2, 3]), Tree([0, 1])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-4, -1)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_complete_nodes_at_level(-3, -2)
[Tree([6]), Tree([4, 5]), Tree([2, 3]), Tree([0, 1])]
```

Trim first node if necessary.

Return list of nodes.

Tree.**get_next_n_nodes_at_level**(*n*, *level*)

Get next *n* nodes at *level* from node.

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Get next 4 nodes at level 2:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, 2)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level 1:

```
>>> tree[0][0].get_next_n_nodes_at_level(3, 1)
[Tree([1]), Tree([2, 3]), Tree([4, 5])]
```

Get next node at level 0:

```
>>> tree[0][0].get_next_n_nodes_at_level(1, 0)
[Tree([[1], [2, 3], [4, 5], [6, 7]])]
```

Get next 4 nodes at level -1:

```
>>> tree[0][0].get_next_n_nodes_at_level(4, -1)
[Tree(1), Tree(2), Tree(3), Tree(4)]
```

Get next 3 nodes at level -2:

```
>>> tree[0][0].get_next_n_nodes_at_level(3, -2)
[Tree([1]), Tree([2, 3]), Tree([4, 5])]
```

Get previous 4 nodes at level 2:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-4, 2)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level 1:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-3, 1)
[Tree([6]), Tree([4, 5]), Tree([2, 3])]
```

Get previous node at level 0:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-1, 0)
[Tree([[0, 1], [2, 3], [4, 5], [6]])]
```

Get previous 4 nodes at level -1:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-4, -1)
[Tree(6), Tree(5), Tree(4), Tree(3)]
```

Get previous 3 nodes at level -2:

```
>>> tree[-1][-1].get_next_n_nodes_at_level(-3, -2)
[Tree([6]), Tree([4, 5]), Tree([2, 3])]
```

Trim first node if necessary.

Return list of nodes.

Tree.**get_node_at_position**(*position*)

Get node at *position*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree.get_node_at_position((2, 1))
Tree(5)
```

Return node.

Tree.**get_position_of_descendant**(*descendant*)

Get position of *descendent* relative to node rather than relative to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[3].get_position_of_descendant(tree[3][0])
(0,)
```

Return tuple of zero or more nonnegative integers.

`Tree.index(node)`
Index of *node*:

```
>>> sequence = [0, 1, 2, 2, 3, 4]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for node in tree:
...     node, tree.index(node)
(Tree(0), 0)
(Tree(1), 1)
(Tree(2), 2)
(Tree(2), 3)
(Tree(3), 4)
(Tree(4), 5)
```

Return nonnegative integer.

`Tree.is_at_level(level)`
True when node is at *level* in containing tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[1][1].is_at_level(-1)
True
```

False otherwise:

```
>>> tree[1][1].is_at_level(0)
False
```

Works for positive, negative and zero-valued *level*.

Return boolean.

`Tree.iterate_at_level(level, reverse=False)`
Iterate tree at *level*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

Left-to-right examples:

```
>>> for x in tree.iterate_at_level(0): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1): x
...
Tree([0, 1])
Tree([2, 3])
Tree([4, 5])
Tree([6, 7])
```

```
>>> for x in tree.iterate_at_level(2): x
...
Tree(0)
Tree(1)
Tree(2)
Tree(3)
Tree(4)
Tree(5)
Tree(6)
Tree(7)
```

```
>>> for x in tree.iterate_at_level(-1): x
...
Tree(0)
Tree(1)
Tree(2)
Tree(3)
Tree(4)
Tree(5)
Tree(6)
Tree(7)
```

```
>>> for x in tree.iterate_at_level(-2): x
...
Tree([0, 1])
Tree([2, 3])
Tree([4, 5])
Tree([6, 7])
```

```
>>> for x in tree.iterate_at_level(-3): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Right-to-left examples:

```
>>> for x in tree.iterate_at_level(0, reverse=True): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> for x in tree.iterate_at_level(1, reverse=True): x
...
Tree([6, 7])
Tree([4, 5])
Tree([2, 3])
Tree([0, 1])
```

```
>>> for x in tree.iterate_at_level(2, reverse=True): x
...
Tree(7)
Tree(6)
Tree(5)
Tree(4)
Tree(3)
Tree(2)
Tree(1)
Tree(0)
```

```
>>> for x in tree.iterate_at_level(-1, reverse=True): x
...
Tree(7)
Tree(6)
Tree(5)
Tree(4)
Tree(3)
Tree(2)
Tree(1)
Tree(0)
```

```
>>> for x in tree.iterate_at_level(-2, reverse=True): x
...
Tree([6, 7])
Tree([4, 5])
Tree([2, 3])
Tree([0, 1])
```

```
>>> for x in tree.iterate_at_level(-3, reverse=True): x
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Return node generator.

Tree.**iterate_depth_first**(*reverse=False*)

Iterate tree depth-first:

Example 1. Iterate tree depth-first from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for node in tree.iterate_depth_first(): node
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
Tree([0, 1])
Tree(0)
Tree(1)
Tree([2, 3])
Tree(2)
Tree(3)
Tree([4, 5])
Tree(4)
Tree(5)
Tree([6, 7])
Tree(6)
Tree(7)
```

Example 2. Iterate tree depth-first from right to left:

```
>>> for node in tree.iterate_depth_first(reverse=True): node
...
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
Tree([6, 7])
Tree(7)
Tree(6)
Tree([4, 5])
Tree(5)
Tree(4)
Tree([2, 3])
Tree(3)
Tree(2)
Tree([0, 1])
Tree(1)
Tree(0)
```

Return node generator.

Tree.**iterate_payload**(*reverse=False*)

Iterate tree payload:

Example 1. Iterate payload from left to right:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for element in tree.iterate_payload():
...     element
...
0
1
2
3
4
5
6
7
```

Example 2. Iterate payload from right to left:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> for element in tree.iterate_payload(reverse=True):
...     element
```

```
...
7
6
5
4
3
2
1
0
```

Return payload generator.

`Tree.`**`remove_node`**`(`*node*`)`

Remove *node* from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree.remove_node(tree[1])
```

```
>>> tree
Tree([[0, 1], [4, 5], [6, 7]])
```

Return none.

`Tree.`**`remove_to_root`**`(`*reverse=False*`)`

Remove node and all nodes left of node to root:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[0][0].remove_to_root()
>>> tree
Tree([[1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[0][1].remove_to_root()
>>> tree
Tree([[2, 3], [4, 5], [6, 7]])
```

```
>>> tree = sequencetools.Tree(sequence)
>>> tree[1].remove_to_root()
>>> tree
Tree([[4, 5], [6, 7]])
```

Modify in-place to root.

Return none.

`Tree.`**`to_nested_lists`**`()`

Change tree to nested lists:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

```
>>> tree.to_nested_lists()
[[0, 1], [2, 3], [4, 5], [6, 7]]
```

Return list of lists.

Special methods

`Tree.`**`__contains__`**`(`*expr*`)`

True when tree contains *expr*:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[-1] in tree
True
```

Otherwise false:

```
>>> tree[-1][-1] in tree
False
```

Return boolean.

`Tree.`**`__eq__`**(*expr*)

True when *expr* is the same type as tree and when the payload of all subtrees are equal:

```
>>> sequence_1 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_1 = sequencetools.Tree(sequence_1)
>>> sequence_2 = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree_2 = sequencetools.Tree(sequence_2)
>>> sequence_3 = [[0, 1], [2, 3], [4, 5]]
>>> tree_3 = sequencetools.Tree(sequence_3)
```

```
>>> tree_1 == tree_1
True
>>> tree_1 == tree_2
True
>>> tree_1 == tree_3
False
>>> tree_2 == tree_1
True
>>> tree_2 == tree_2
True
>>> tree_2 == tree_3
False
>>> tree_3 == tree_1
False
>>> tree_3 == tree_2
False
>>> tree_3 == tree_3
True
```

Return boolean.

`Tree.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`Tree.`**`__getitem__`**(*expr*)

Get item from tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree[-1]
Tree([6, 7])
```

Get slice from tree:

```
>>> tree[-2:]
[Tree([4, 5]), Tree([6, 7])]
```

Return node.

`Tree.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

Tree.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Tree.**__len__**()
Return the number of children in tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> len(tree)
4
```

Return nonnegative integer.

Tree.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Tree.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Tree.**__repr__**()
Interpreter representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> tree
Tree([[0, 1], [2, 3], [4, 5], [6, 7]])
```

Return string.

Tree.**__str__**()
String representation of tree:

```
>>> sequence = [[0, 1], [2, 3], [4, 5], [6, 7]]
>>> tree = sequencetools.Tree(sequence)
```

```
>>> str(tree)
'[[0, 1], [2, 3], [4, 5], [6, 7]]'
```

Return string.

30.2 Functions

30.2.1 sequencetools.all_are_assignable_integers

sequencetools.**all_are_assignable_integers**(*expr*)
New in version 2.0. True when *expr* is a sequence and all elements in *expr* are notehead-assignable integers:

```
>>> sequencetools.all_are_assignable_integers([1, 2, 3, 4, 6, 7, 8, 12, 14, 15, 16])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_assignable_integers([])
True
```

False otherwise:

```
>>> sequencetools.all_are_assignable_integers('foo')
False
```

Return boolean.

30.2.2 sequencetools.all_are_equal

sequencetools.**all_are_equal**(*expr*)

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are equal:

```
>>> sequencetools.all_are_equal([99, 99, 99, 99, 99, 99])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_equal([])
True
```

False otherwise:

```
>>> sequencetools.all_are_equal(17)
False
```

Return boolean.

30.2.3 sequencetools.all_are_integer_equivalent_exprs

sequencetools.**all_are_integer_equivalent_exprs**(*expr*)

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are integer-equivalent expressions:

```
>>> sequencetools.all_are_integer_equivalent_exprs([1, '2', 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> sequencetools.all_are_integer_equivalent_exprs([1, '2', 3.5, 4])
False
```

Return boolean.

30.2.4 sequencetools.all_are_integer_equivalent_numbers

sequencetools.**all_are_integer_equivalent_numbers**(*expr*)

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are integer-equivalent numbers:

```
>>> sequencetools.all_are_integer_equivalent_numbers([1, 2, 3.0, Fraction(4, 1)])
True
```

Otherwise false:

```
>>> sequencetools.all_are_integer_equivalent_numbers([1, 2, 3.5, 4])
False
```

Return boolean.

30.2.5 sequencetools.all_are_nonnegative_integer_equivalent_numbers

sequencetools.**all_are_nonnegative_integer_equivalent_numbers**(*expr*)

New in version 2.0. True *expr* is a sequence and when all elements in *expr* are nonnegative integer-equivalent numbers. Otherwise false:

```
>>> expr = [0, 0.0, Fraction(0), 2, 2.0, Fraction(2)]
>>> sequencetools.all_are_nonnegative_integer_equivalent_numbers(expr)
True
```

Return boolean.

30.2.6 sequencetools.all_are_nonnegative_integer_powers_of_two

sequencetools.**all_are_nonnegative_integer_powers_of_two**(*expr*)

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are nonnegative integer powers of two:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two([0, 1, 1, 1, 2, 4, 32, 32])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two([])
True
```

False otherwise:

```
>>> sequencetools.all_are_nonnegative_integer_powers_of_two(17)
False
```

Return boolean.

30.2.7 sequencetools.all_are_nonnegative_integers

sequencetools.**all_are_nonnegative_integers**(*expr*)

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are nonnegative integers:

```
>>> sequencetools.all_are_nonnegative_integers([0, 1, 2, 99])
True
```

Otherwise false:

```
>>> sequencetools.all_are_nonnegative_integers([0, 1, 2, -99])
False
```

Return boolean.

30.2.8 sequencetools.all_are_numbers

sequencetools.**all_are_numbers**(*expr*)

New in version 1.1. True when *expr* is a sequence and all elements in *expr* are numbers:

```
>>> sequencetools.all_are_numbers([1, 2, 3.0, Fraction(13, 8)])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_numbers([])
True
```

False otherwise:

```
>>> sequencetools.all_are_numbers(17)
False
```

Return boolean.

30.2.9 sequencetools.all_are_pairs

`sequencetools.all_are_pairs(expr)`

True when *expr* is a sequence whose members are all sequences of length 2:

```
>>> sequencetools.all_are_pairs([(1, 2), (3, 4), (5, 6), (7, 8)])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_pairs([])
True
```

False otherwise:

```
>>> sequencetools.all_are_pairs('foo')
False
```

Return boolean.

30.2.10 sequencetools.all_are_pairs_of_types

`sequencetools.all_are_pairs_of_types(expr, first_type, second_type)`

True when *expr* is a sequence whose members are all sequences of length 2, and where the first member of each pair is an instance of *first_type* and where the second member of each pair is an instance of *second_type*:

```
>>> sequencetools.all_are_pairs_of_types([(1., 'a'), (2.1, 'b'), (3.45, 'c')], float, str)
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_pairs_of_types([], float, str)
True
```

False otherwise:

```
>>> sequencetools.all_are_pairs_of_types('foo', float, str)
False
```

Return boolean.

30.2.11 sequencetools.all_are_positive_integer_equivalent_numbers

`sequencetools.all_are_positive_integer_equivalent_numbers(expr)`

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are positive integer-equivalent numbers. Otherwise false:

```
>>> sequencetools.all_are_positive_integer_equivalent_numbers([Fraction(4, 2), 2.0, 2])
True
```

Return boolean.

30.2.12 sequencetools.all_are_positive_integers

`sequencetools.all_are_positive_integers(expr)`

New in version 2.0. True when *expr* is a sequence and all elements in *expr* are positive integers:

```
>>> sequencetools.all_are_positive_integers([1, 2, 3, 99])
True
```

Otherwise false:

```
>>> sequencetools.all_are_positive_integers(17)
False
```

Return boolean.

30.2.13 sequencetools.all_are_unequal

sequencetools.**all_are_unequal**(*expr*)

New in version 1.1. True when *expr* is a sequence all elements in *expr* are unequal:

```
>>> sequencetools.all_are_unequal([1, 2, 3, 4, 9])
True
```

True when *expr* is an empty sequence:

```
>>> sequencetools.all_are_unequal([])
True
```

False otherwise:

```
>>> sequencetools.all_are_unequal(17)
False
```

Return boolean.

30.2.14 sequencetools.count_length_two_runs_in_sequence

sequencetools.**count_length_two_runs_in_sequence**(*sequence*)

New in version 1.1. Count length-2 runs in *sequence*:

```
>>> sequencetools.count_length_two_runs_in_sequence([0, 0, 1, 1, 1, 2, 3, 4, 5])
3
```

Return nonnegative integer.

30.2.15 sequencetools.divide_sequence_elements_by_greatest_common_divisor

sequencetools.**divide_sequence_elements_by_greatest_common_divisor**(*sequence*)

New in version 2.0. Divide *sequence* elements by greatest common divisor:

```
>>> sequencetools.divide_sequence_elements_by_greatest_common_divisor([2, 2, -8, -16])
[1, 1, -4, -8]
```

Allow negative *sequence* elements.

Raise type error on noninteger *sequence* elements.

Raise not implemented error when `0` in *sequence*.

Return new *sequence* object.

30.2.16 sequencetools.flatten_sequence

sequencetools.**flatten_sequence**(*sequence*, *klasses=None*, *depth=-1*)

New in version 1.1. Flatten *sequence*:

```
>>> sequencetools.flatten_sequence([1, [2, 3, [4]], 5, [6, 7, [8]]])
[1, 2, 3, 4, 5, 6, 7, 8]
```

Flatten *sequence* to depth `1`:

```
>>> sequencetools.flatten_sequence([1, [2, 3, [4]], 5, [6, 7, [8]]], depth=1)
[1, 2, 3, [4], 5, 6, 7, [8]]
```

Flatten *sequence* to depth 2:

```
>>> sequencetools.flatten_sequence([1, [2, 3, [4]], 5, [6, 7, [8]]], depth=2)
[1, 2, 3, 4, 5, 6, 7, 8]
```

Leave *sequence* unchanged.

Return newly constructed *sequence* object.

30.2.17 sequencetools.flatten_sequence_at_indices

sequencetools.**flatten_sequence_at_indices**(*sequence*, *indices*, *klasses=None*, *depth=-1*)

New in version 2.0. Flatten *sequence* at *indices*:

```
>>> sequencetools.flatten_sequence_at_indices([0, 1, [2, 3, 4], [5, 6, 7]], [3])
[0, 1, [2, 3, 4], 5, 6, 7]
```

Flatten *sequence* at negative *indices*:

```
>>> sequencetools.flatten_sequence_at_indices([0, 1, [2, 3, 4], [5, 6, 7]], [-1])
[0, 1, [2, 3, 4], 5, 6, 7]
```

Leave *sequence* unchanged.

Return newly constructed *sequence* object.

30.2.18 sequencetools.get_indices_of_sequence_elements_equal_to_true

sequencetools.**get_indices_of_sequence_elements_equal_to_true**(*sequence*)

New in version 1.1. Get indices of *sequence* elements equal to true:

```
>>> sequence = [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1]
```

```
>>> sequencetools.get_indices_of_sequence_elements_equal_to_true(sequence)
(3, 4, 5, 9, 10, 11, 12)
```

Return newly constructed tuple of zero or more nonnegative integers.

30.2.19 sequencetools.get_sequence_degree_of_rotational_symmetry

sequencetools.**get_sequence_degree_of_rotational_symmetry**(*sequence*)

New in version 2.0. Change *sequence* to degree of rotational symmetry:

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 3, 4, 5, 6])
1
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 3, 1, 2, 3])
2
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 2, 1, 2, 1, 2])
3
```

```
>>> sequencetools.get_sequence_degree_of_rotational_symmetry([1, 1, 1, 1, 1, 1])
6
```

Return positive integer.

30.2.20 sequencetools.get_sequence_element_at_cyclic_index

sequencetools.**get_sequence_element_at_cyclic_index**(*sequence*, *index*)
New in version 2.0. Get *sequence* element at nonnegative cyclic *index*:

```
>>> for index in range(10):
...     print '%s\t%s' % (index, sequencetools.get_sequence_element_at_cyclic_index(
...     'string', index))
...
0   s
1   t
2   r
3   i
4   n
5   g
6   s
7   t
8   r
9   i
```

Get *sequence* element at negative cyclic *index*:

```
>>> for index in range(1, 11):
...     print '%s\t%s' % (-index, sequencetools.get_sequence_element_at_cyclic_index(
...     'string', -index))
...
-1    g
-2    n
-3    i
-4    r
-5    t
-6    s
-7    g
-8    n
-9    i
-10   r
```

Return reference to *sequence* element.

30.2.21 sequencetools.get_sequence_elements_at_indices

sequencetools.**get_sequence_elements_at_indices**(*sequence*, *indices*)
New in version 2.0. Get *sequence* elements at *indices*:

```
>>> sequencetools.get_sequence_elements_at_indices('string of text', (2, 3, 10, 12))
('r', 'i', 't', 'x')
```

Return newly constructed tuple of references to *sequence* elements.

30.2.22 sequencetools.get_sequence_elements_frequency_distribution

sequencetools.**get_sequence_elements_frequency_distribution**(*sequence*)
New in version 2.0. Get *sequence* elements frequency distribution:

```
>>> sequence = [1, 3, 3, 3, 2, 1, 1, 2, 3, 3, 1, 2]
```

```
>>> sequencetools.get_sequence_elements_frequency_distribution(sequence)
[(1, 4), (2, 3), (3, 5)]
```

Return list of element / count pairs.

30.2.23 sequencetools.get_sequence_period_of_rotation

sequencetools.**get_sequence_period_of_rotation**(*sequence*, *n*)
New in version 2.0. Change *sequence* to period of rotation:

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 1)
3
```

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 2)
3
```

```
>>> sequencetools.get_sequence_period_of_rotation([1, 2, 3, 1, 2, 3], 3)
1
```

Return positive integer.

30.2.24 sequencetools.increase_sequence_elements_at_indices_by_addenda

sequencetools.**increase_sequence_elements_at_indices_by_addenda**(*sequence*, *addenda*, *indices*)

New in version 1.1. Increase *sequence* by *addenda* at *indices*:

```
>>> sequence = [1, 1, 2, 3, 5, 5, 1, 2, 5, 5, 6]
```

```
>>> sequencetools.increase_sequence_elements_at_indices_by_addenda(
...     sequence, [0.5, 0.5], [0, 4, 8])
[1.5, 1.5, 2, 3, 5.5, 5.5, 1, 2, 5.5, 5.5, 6]
```

Return list.

30.2.25 sequencetools.increase_sequence_elements_cyclically_by_addenda

sequencetools.**increase_sequence_elements_cyclically_by_addenda**(*sequence*, *addenda*, *shield=True*)

New in version 1.1.. Increase *sequence* cyclically by *addenda*:

```
>>> sequencetools.increase_sequence_elements_cyclically_by_addenda(
...     range(10), [10, -10], shield=False)
[10, -9, 12, -7, 14, -5, 16, -3, 18, -1]
```

Increase *sequence* cyclically by addenda and map nonpositive values to `1`:

```
>>> sequencetools.increase_sequence_elements_cyclically_by_addenda(
...         range(10), [10, -10], shield=True)
[10, 1, 12, 1, 14, 1, 16, 1, 18, 1]
```

Return list.

30.2.26 sequencetools.interlace_sequences

sequencetools.**interlace_sequences**(**sequences*)

New in version 1.1. Interlace *sequences*:

```
>>> k = range(100, 103)
>>> l = range(200, 201)
>>> m = range(300, 303)
>>> n = range(400, 408)
>>> sequencetools.interlace_sequences(k, l, m, n)
[100, 200, 300, 400, 101, 301, 401, 102, 302, 402, 403, 404, 405, 406, 407]
```

Return list.

30.2.27 sequencetools.is_fraction_equivalent_pair

sequencetools.**is_fraction_equivalent_pair**(*expr*)

New in version 2.9. True when *expr* is an integer-equivalent pair of numbers excluding 0 as the second term:

```
>>> sequencetools.is_fraction_equivalent_pair((2, 3))
True
```

Otherwise false:

```
>>> sequencetools.is_fraction_equivalent_pair((2, 0))
False
```

Return boolean.

30.2.28 sequencetools.is_integer_equivalent_n_tuple

sequencetools.**is_integer_equivalent_n_tuple**(*expr*, *n*)

New in version 2.9. True when *expr* is a tuple of *n* integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_n_tuple((2.0, '3', Fraction(4, 1)), 3)
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_n_tuple((2.5, '3', Fraction(4, 1)), 3)
False
```

Return boolean.

30.2.29 sequencetools.is_integer_equivalent_pair

sequencetools.**is_integer_equivalent_pair**(*expr*)

New in version 2.9. True when *expr* is a pair of integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_pair((2.0, '3'))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_pair((2.5, '3'))
False
```

Return boolean.

30.2.30 sequencetools.is_integer_equivalent_singleton

sequencetools.**is_integer_equivalent_singleton**(*expr*)

New in version 2.9. True when *expr* is a singleton of integer-equivalent expressions:

```
>>> sequencetools.is_integer_equivalent_singleton((2.0,))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_equivalent_singleton((2.5,))
False
```

Return boolean.

30.2.31 sequencetools.is_integer_n_tuple

sequencetools.**is_integer_n_tuple**(*expr*, *n*)

New in version 2.9. True when *expr* is an integer tuple of length *n*:

```
>>> sequencetools.is_integer_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> sequencetools.is_integer_n_tuple((19, 20, 'text'), 3)
False
```

Return boolean.

30.2.32 sequencetools.is_integer_pair

sequencetools.**is_integer_pair**(*expr*)

New in version 2.9. True when *expr* is an integer tuple of length 2:

```
>>> sequencetools.is_integer_pair((19, 20))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_pair(('some', 'text'))
False
```

Return boolean.

30.2.33 sequencetools.is_integer_singleton

sequencetools.**is_integer_singleton**(*expr*)

New in version 2.9. True when *expr* is an integer tuple of of length 1:

```
>>> sequencetools.is_integer_singleton((19,))
True
```

Otherwise false:

```
>>> sequencetools.is_integer_singleton(('text',))
False
```

Return boolean.

30.2.34 sequencetools.is_monotonically_decreasing_sequence

sequencetools.**is_monotonically_decreasing_sequence**(*expr*)

New in version 2.0. True when *expr* is a sequence and the elements in *expr* decrease monotonically:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not decrease monotonically:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
False
```

True when *expr* is a sequence and *expr* is empty:

```
>>> expr = []
>>> sequencetools.is_monotonically_decreasing_sequence(expr)
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_monotonically_decreasing_sequence(17)
False
```

Return boolean.

30.2.35 sequencetools.is_monotonically_increasing_sequence

sequencetools.**is_monotonically_increasing_sequence**(*expr*)

New in version 2.0. True when *expr* is a sequence and the elements in *expr* increase monotonically:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not increase monotonically:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_monotonically_increasing_sequence(expr)
False
```

True when *expr* is a sequence and *expr* is empty:

```
>>> expr = []
>>> sequencetools.is_monotonically_increasing_sequence(expr)
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_monotonically_increasing_sequence(17)
False
```

Return boolean.

30.2.36 sequencetools.is_n_tuple

sequencetools.**is_n_tuple**(*expr*, *n*)

True when *expr* is a tuple of length *n*:

```
>>> sequencetools.is_n_tuple((19, 20, 21), 3)
True
```

Otherwise false:

```
>>> sequencetools.is_n_tuple((19, 20, 21), 4)
False
```

Return boolean.

30.2.37 sequencetools.is_null_tuple

`sequencetools.is_null_tuple(expr)`

New in version 2.9. True when *expr* is a tuple of length `0`:

```
>>> sequencetools.is_null_tuple(())
True
```

Otherwise false:

```
>>> sequencetools.is_null_tuple((19, 20, 21))
False
```

Return boolean.

30.2.38 sequencetools.is_pair

`sequencetools.is_pair(expr)`

New in version 2.9. True when *expr* is a tuple of length `2`:

```
>>> sequencetools.is_pair((19, 20))
True
```

Otherwise false:

```
>>> sequencetools.is_pair((19, 20, 21))
False
```

Return boolean.

30.2.39 sequencetools.is_permutation

`sequencetools.is_permutation(expr, length=None)`

New in version 2.0. True when *expr* is a permutation:

```
>>> sequencetools.is_permutation([4, 5, 0, 3, 2, 1])
True
```

Otherwise false:

```
>>> sequencetools.is_permutation([1, 1, 5, 3, 2, 1])
False
```

True when *expr* is a permutation of first *length* nonnegative integers:

```
>>> sequencetools.is_permutation([4, 5, 0, 3, 2, 1], length=6)
True
```

Otherwise false:

```
>>> sequencetools.is_permutation([4, 0, 3, 2, 1], length=6)
False
```

Return boolean.

30.2.40 sequencetools.is_repetition_free_sequence

`sequencetools.is_repetition_free_sequence(expr)`

New in version 2.0. True when *expr* is a sequence and *expr* is repetition free:

```
>>> sequencetools.is_repetition_free_sequence([0, 1, 2, 6, 7, 8])
True
```

False when *expr* is a sequence and *expr* is not repetition free:

```
>>> sequencetools.is_repetition_free_sequence([0, 1, 2, 2, 7, 8])
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_repetition_free_sequence([])
True
```

False *expr* is not a sequence:

```
>>> sequencetools.is_repetition_free_sequence(17)
False
```

Return boolean.

30.2.41 sequencetools.is_restricted_growth_function

`sequencetools.is_restricted_growth_function(expr)`

New in version 2.0. True when *expr* is a sequence and *expr* meets the criteria for a restricted growth function:

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 1])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 2])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 2, 1])
True
```

```
>>> sequencetools.is_restricted_growth_function([1, 1, 2, 2])
True
```

Otherwise false:

```
>>> sequencetools.is_restricted_growth_function([1, 1, 1, 3])
False
```

```
>>> sequencetools.is_restricted_growth_function(17)
False
```

A restricted growth function is a sequence `l` such that `l[0] == 1` and such that `l[i] <= max(l[:i]) + 1` for `1 <= i <= len(l)`.

Return boolean.

30.2.42 sequencetools.is_singleton

`sequencetools.is_singleton(expr)`

New in version 2.9. True when *expr* is a tuple of length `1`:

```
>>> sequencetools.is_singleton((19,))
True
```

Otherwise false:

```
>>> sequencetools.is_singleton((19, 20, 21))
False
```

Return boolean.

30.2.43 sequencetools.is_strictly_decreasing_sequence

sequencetools.**is_strictly_decreasing_sequence**(*expr*)
New in version 2.0. True when *expr* is a sequence and the elements in *expr* decrease strictly:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not decrease strictly:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_strictly_decreasing_sequence(expr)
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_strictly_decreasing_sequence([])
True
```

False *expr* is not a sequence:

```
>>> sequencetools.is_strictly_decreasing_sequence(17)
False
```

Return boolean.

30.2.44 sequencetools.is_strictly_increasing_sequence

sequencetools.**is_strictly_increasing_sequence**(*expr*)
New in version 2.0. True when *expr* is a sequence and the elements in *expr* increase strictly:

```
>>> expr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sequencetools.is_strictly_increasing_sequence(expr)
True
```

False when *expr* is a sequence and the elements in *expr* do not increase strictly:

```
>>> expr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 2, 1, 0]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

```
>>> expr = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3]
>>> sequencetools.is_strictly_increasing_sequence(expr)
False
```

True when *expr* is an empty sequence:

```
>>> sequencetools.is_strictly_increasing_sequence([])
True
```

False when *expr* is not a sequence:

```
>>> sequencetools.is_strictly_increasing_sequence(17)
False
```

Return boolean.

30.2.45 sequencetools.iterate_sequence_cyclically

sequencetools.**iterate_sequence_cyclically**(*sequence*, *step=1*, *start=0*, *length='inf'*)

New in version 1.1. Iterate *sequence* cyclically according to *step*, *start* and *length*:

```
>>> sequence = [1, 2, 3, 4, 5, 6, 7]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, length=20))
[1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, 2, length=20))
[1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, 2, 3, length=20))
[4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7, 2, 4, 6, 1, 3, 5, 7]
```

```
>>> list(sequencetools.iterate_sequence_cyclically(sequence, -2, 5, length=20))
[6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3]
```

Allow generator input:

```
>>> list(sequencetools.iterate_sequence_cyclically(xrange(1, 8), -2, 5, length=20))
[6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3, 1, 6, 4, 2, 7, 5, 3]
```

Set *step* to jump size and direction across sequence.

Set *start* to the index of *sequence* where the function begins iterating.

Set *length* to number of elements to return. Set to `'inf'` to return infinitely.

Return generator.

30.2.46 sequencetools.iterate_sequence_cyclically_from_start_to_stop

sequencetools.**iterate_sequence_cyclically_from_start_to_stop**(*sequence*, *start*, *stop*)

New in version 1.1. Iterate *sequence* cyclically from *start* to *stop*:

```
>>> list(sequencetools.iterate_sequence_cyclically_from_start_to_stop(range(20), 18, 10))
[18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Return generator of references to *sequence* elements.

30.2.47 sequencetools.iterate_sequence_forward_and_backward_nonoverlapping

sequencetools.**iterate_sequence_forward_and_backward_nonoverlapping**(*sequence*)

New in version 2.0. Iterate *sequence* first forward and then backward, with first and last elements repeated:

```
>>> list(sequencetools.iterate_sequence_forward_and_backward_nonoverlapping(
...      [1, 2, 3, 4, 5]))
[1, 2, 3, 4, 5, 5, 4, 3, 2, 1]
```

Return generator.

30.2.48 sequencetools.iterate_sequence_forward_and_backward_overlapping

sequencetools.**iterate_sequence_forward_and_backward_overlapping**(*sequence*)
New in version 2.0. Iterate *sequence* first forward and then backward, with first and last elements appearing only once:

```
>>> list(sequencetools.iterate_sequence_forward_and_backward_overlapping([1, 2, 3, 4, 5]))
[1, 2, 3, 4, 5, 4, 3, 2]
```

Return generator.

30.2.49 sequencetools.iterate_sequence_nwise_cyclic

sequencetools.**iterate_sequence_nwise_cyclic**(*sequence*, *n*)
New in version 2.0. Iterate elements in *sequence* cyclically *n* at a time:

```
>>> g = sequencetools.iterate_sequence_nwise_cyclic(range(6), 3)
>>> for n in range(10):
...    print g.next()
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(4, 5, 0)
(5, 0, 1)
(0, 1, 2)
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
```

Return generator.

30.2.50 sequencetools.iterate_sequence_nwise_strict

sequencetools.**iterate_sequence_nwise_strict**(*sequence*, *n*)
New in version 2.0. Iterate elements in *sequence* *n* at a time:

```
>>> for x in sequencetools.iterate_sequence_nwise_strict(range(10), 4):
...      x
...
(0, 1, 2, 3)
(1, 2, 3, 4)
(2, 3, 4, 5)
(3, 4, 5, 6)
(4, 5, 6, 7)
(5, 6, 7, 8)
(6, 7, 8, 9)
```

Return generator.

30.2.51 sequencetools.iterate_sequence_nwise_wrapped

sequencetools.**iterate_sequence_nwise_wrapped**(*sequence*, *n*)
New in version 2.0. Iterate elements in *sequence* *n* at a time wrapped to beginning:

```
>>> list(sequencetools.iterate_sequence_nwise_wrapped(range(6), 3))
[(0, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, 5), (4, 5, 0), (5, 0, 1)]
```

Return generator.

30.2.52 sequencetools.iterate_sequence_pairwise_cyclic

sequencetools.**iterate_sequence_pairwise_cyclic**(*sequence*)
New in version 1.1. Iterate *sequence* pairwise cyclic:

```
>>> generator = sequencetools.iterate_sequence_pairwise_cyclic(range(6))
```

```
>>> generator.next()
(0, 1)
>>> generator.next()
(1, 2)
>>> generator.next()
(2, 3)
>>> generator.next()
(3, 4)
>>> generator.next()
(4, 5)
>>> generator.next()
(5, 0)
>>> generator.next()
(0, 1)
>>> generator.next()
(1, 2)
```

Return pair generator.

30.2.53 sequencetools.iterate_sequence_pairwise_strict

sequencetools.**iterate_sequence_pairwise_strict**(*sequence*)
New in version 1.1. Iterate *sequence* pairwise strict:

```
>>> list(sequencetools.iterate_sequence_pairwise_strict(range(6)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]
```

Return pair generator.

30.2.54 sequencetools.iterate_sequence_pairwise_wrapped

sequencetools.**iterate_sequence_pairwise_wrapped**(*sequence*)
New in version 1.1. Iterate *sequence* pairwise wrapped:

```
>>> list(sequencetools.iterate_sequence_pairwise_wrapped(range(6)))
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 0)]
```

Return pair generator.

30.2.55 sequencetools.join_subsequences

sequencetools.**join_subsequences**(*sequence*)
New in version 2.4. Join subsequences in *sequence*:

```
>>> sequencetools.join_subsequences([(1, 2, 3), (), (4, 5), (), (6,)])
(1, 2, 3, 4, 5, 6)
```

Return newly constructed object of subsequence type.

30.2.56 sequencetools.join_subsequences_by_sign_of_subsequence_elements

sequencetools.**join_subsequences_by_sign_of_subsequence_elements**(*sequence*)

New in version 1.1. Join subsequences in *sequence* by sign:

```
>>> sequence = [[1, 2], [3, 4], [-5, -6, -7], [-8, -9, -10], [11, 12]]
>>> sequencetools.join_subsequences_by_sign_of_subsequence_elements(sequence)
[[1, 2, 3, 4], [-5, -6, -7, -8, -9, -10], [11, 12]]
```

```
>>> sequence = [[1, 2], [], [], [3, 4, 5], [6, 7]]
>>> sequencetools.join_subsequences_by_sign_of_subsequence_elements(sequence)
[[1, 2], [], [3, 4, 5, 6, 7]]
```

Return new list.

30.2.57 sequencetools.map_sequence_elements_to_canonic_tuples

sequencetools.**map_sequence_elements_to_canonic_tuples**(*sequence*, *decrease_parts_monotonically=True*)

New in version 1.1. Partition *sequence* elements into canonic parts that decrease monotonically:

```
>>> sequencetools.map_sequence_elements_to_canonic_tuples(
...     range(10))
[(0,), (1,), (2,), (3,), (4,), (4, 1), (6,), (7,), (8,), (8, 1)]
```

Partition *sequence* elements into canonic parts that increase monotonically:

```
>>> sequencetools.map_sequence_elements_to_canonic_tuples(
...     range(10), decrease_parts_monotonically=False)
[(0,), (1,), (2,), (3,), (4,), (1, 4), (6,), (7,), (8,), (1, 8)]
```

Raise type error when *sequence* is not a list.

Raise value error on noninteger elements in *sequence*.

Return list of tuples.

30.2.58 sequencetools.map_sequence_elements_to_numbered_sublists

sequencetools.**map_sequence_elements_to_numbered_sublists**(*sequence*)

New in version 1.1. Map *sequence* elements to numbered sublists:

```
>>> sequencetools.map_sequence_elements_to_numbered_sublists([1, 2, -3, -4, 5])
[[1], [2, 3], [-4, -5, -6], [-7, -8, -9, -10], [11, 12, 13, 14, 15]]
```

```
>>> sequencetools.map_sequence_elements_to_numbered_sublists([1, 0, -3, -4, 5])
[[1], [], [-2, -3, -4], [-5, -6, -7, -8], [9, 10, 11, 12, 13]]
```

Note that numbering starts at `1`.

Return newly constructed list of lists.

30.2.59 sequencetools.merge_duration_sequences

sequencetools.**merge_duration_sequences**(**sequences*)

Merge duration *sequences*:

```
>>> sequencetools.merge_duration_sequences([10, 10, 10], [7])
[7, 3, 10, 10]
```

Merge more duration sequences:

```
>>> sequencetools.merge_duration_sequences([10, 10, 10], [10, 10])
[10, 10, 10]
```

The idea is that each sequence element represents a duration.

Return list.

30.2.60 sequencetools.negate_absolute_value_of_sequence_elements_at_indices

sequencetools.**negate_absolute_value_of_sequence_elements_at_indices**(*sequence*, *indices*)

New in version 1.1. Negate the absolute value of *sequence* elements at *indices*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_at_indices(sequence, [0, 1, 2])
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Return newly constructed list.

30.2.61 sequencetools.negate_absolute_value_of_sequence_elements_cyclically

sequencetools.**negate_absolute_value_of_sequence_elements_cyclically**(*sequence*, *indices*, *period*)

New in version 2.0. Negate the absolute value of *sequence* elements at *indices* cyclically according to *period*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_absolute_value_of_sequence_elements_cyclically(
...     sequence, [0, 1, 2], 5)
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Return newly constructed list.

30.2.62 sequencetools.negate_sequence_elements_at_indices

sequencetools.**negate_sequence_elements_at_indices**(*sequence*, *indices*)

New in version 1.1. Negate *sequence* elements at *indices*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_at_indices(sequence, [0, 1, 2])
[-1, -2, -3, 4, 5, -6, -7, -8, -9, -10]
```

Return newly constructed list.

30.2.63 sequencetools.negate_sequence_elements_cyclically

sequencetools.**negate_sequence_elements_cyclically**(*sequence*, *indices*, *period*)

New in version 2.0. Negate *sequence* elements at *indices* cyclically according to *period*:

```
>>> sequence = [1, 2, 3, 4, 5, -6, -7, -8, -9, -10]
```

```
>>> sequencetools.negate_sequence_elements_cyclically(sequence, [0, 1, 2], 5)
[-1, -2, -3, 4, 5, 6, 7, 8, -9, -10]
```

Return newly constructed list.

30.2.64 sequencetools.overwrite_sequence_elements_at_indices

sequencetools.**overwrite_sequence_elements_at_indices**(*sequence*, *pairs*)
New in version 1.1. Overwrite *sequence* elements at indices according to *pairs*:

```
>>> sequencetools.overwrite_sequence_elements_at_indices(range(10), [(0, 3), (5, 3)])
[0, 0, 0, 3, 4, 5, 5, 5, 8, 9]
```

Set *pairs* to a list of `(anchor_index, length)` pairs.

Return new list.

30.2.65 sequencetools.pair_duration_sequence_elements_with_input_pair_values

sequencetools.**pair_duration_sequence_elements_with_input_pair_values**(*duration_sequence*, *input_pairs*)
New in version 2.10. Pair *duration_sequence* elements with the values of *input_pairs*:

```
>>> duration_sequence = [10, 10, 10, 10]
>>> input_pairs = [('red', 1), ('orange', 18), ('yellow', 200)]
```

```
>>> sequencetools.pair_duration_sequence_elements_with_input_pair_values(
... duration_sequence, input_pairs)
[(10, 'red'), (10, 'orange'), (10, 'yellow'), (10, 'yellow')]
```

Return a list of `(element, value)` output pairs.

The *input_pairs* argument must be a list of `(value, duration)` pairs.

The basic idea behind the function is model which input pair value is in effect at the start of each element in *duration_sequence*.

30.2.66 sequencetools.partition_sequence_by_backgrounded_weights

sequencetools.**partition_sequence_by_backgrounded_weights**(*sequence*, *weights*)
New in version 2.9. Partition *sequence* by backgrounded *weights*:

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(
...     [-5, -15, -10], [20, 10])
[[-5, -15], [-10]]
```

Further examples:

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(
...     [-5, -15, -10], [5, 5, 5, 5, 5, 5])
[[-5], [-15], [], [], [-10], []]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(
...     [-5, -15, -10], [1, 29])
[[-5], [-15, -10]]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(
...     [-5, -15, -10], [2, 28])
[[-5], [-15, -10]]
```

```
>>> sequencetools.partition_sequence_by_backgrounded_weights(
...     [-5, -15, -10], [1, 1, 1, 1, 1, 25])
[[-5], [], [], [], [], [-15, -10]]
```

The term *backgrounded* is a short-hand concocted specifically for this function; rely on the formal definition to understand the function actually does.

Input constraint: the weight of *sequence* must equal the weight of *weights* exactly.

The signs of the elements in *sequence* are ignored.

Formal definition: partition *sequence* into *parts* such that (1.) the length of *parts* equals the length of *weights*; (2.) the elements in *sequence* appear in order in *parts*; and (3.) some final condition that is difficult to formalize.

Notionally what's going on here is that the elements of *weights* are acting as a list of successive time intervals into which the elements of *sequence* are being fit in accordance with the start offset of each *sequence* element.

The function models the grouping together of successive timespans according to which of an underlying sequence of time intervals it is in which each time span begins.

Note that, for any input to this function, the flattened output of this function always equals *sequence* exactly.

Note too that while *partition* is being used here in the sense of the other partitioning functions in the API, the distinguishing feature is this funciton is its ability to produce empty lists as output.

Return list of *sequence* objects.

30.2.67 sequencetools.partition_sequence_by_counts

`sequencetools.partition_sequence_by_counts`(*sequence*, *counts*, *cyclic=False*, *overhang=False*, *copy_elements=False*)

New in version 1.1. Example 1a. Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(10), [3], cyclic=False, overhang=False)
[[0, 1, 2]]
```

Example 1b. Partition sequence once by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(16), [4, 3], cyclic=False, overhang=False)
[[0, 1, 2, 3], [4, 5, 6]]
```

Example 2a. Partition sequence cyclically by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(10), [3], cyclic=True, overhang=False)
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

Example 2b. Partition sequence cyclically by counts without overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(16), [4, 3], cyclic=True, overhang=False)
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10], [11, 12, 13]]
```

Example 3a. Partition sequence once by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(10), [3], cyclic=False, overhang=True)
[[0, 1, 2], [3, 4, 5, 6, 7, 8, 9]]
```

Example 3b. Partition sequence once by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(16), [4, 3], cyclic=False, overhang=True)
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10, 11, 12, 13, 14, 15]]
```

Example 4a. Partition sequence cyclically by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(10), [3], cyclic=True, overhang=True)
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

Example 4b. Partition sequence cyclically by counts with overhang:

```
>>> sequencetools.partition_sequence_by_counts(
...     range(16), [4, 3], cyclic=True, overhang=True)
[[0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10], [11, 12, 13], [14, 15]]
```

Return list of sequence objects.

30.2.68 sequencetools.partition_sequence_by_ratio_of_lengths

sequencetools.**partition_sequence_by_ratio_of_lengths**(*sequence*, *lengths*)

New in version 2.0. Partition *sequence* by ratio of *lengths*:

```
>>> sequence = tuple(range(10))
```

```
>>> sequencetools.partition_sequence_by_ratio_of_lengths(sequence, [1, 1, 2])
[(0, 1, 2), (3, 4), (5, 6, 7, 8, 9)]
```

Use rounding magic to avoid fractional part lengths.

Return list of *sequence* objects.

30.2.69 sequencetools.partition_sequence_by_ratio_of_weights

sequencetools.**partition_sequence_by_ratio_of_weights**(*sequence*, *weights*)

New in version 2.0. Partition *sequence* by ratio of *weights*:

```
>>> sequencetools.partition_sequence_by_ratio_of_weights([1] * 10, [1, 1, 1])
[[1, 1, 1], [1, 1, 1, 1], [1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights([1] * 10, [1, 1, 1, 1])
[[1, 1, 1], [1, 1], [1, 1, 1], [1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights([1] * 10, [2, 2, 3])
[[1, 1, 1], [1, 1, 1], [1, 1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights([1] * 10, [3, 2, 2])
[[1, 1, 1, 1], [1, 1, 1], [1, 1, 1]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2], [1, 1])
[[1, 1, 1, 1, 1, 1, 2, 2], [2, 2, 2, 2]]
```

```
>>> sequencetools.partition_sequence_by_ratio_of_weights(
...     [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2], [1, 1, 1])
[[1, 1, 1, 1, 1, 1], [2, 2, 2], [2, 2, 2]]
```

Weights of parts of returned list equal *weights_ratio* proportions with some rounding magic.

Return list of lists.

30.2.70 sequencetools.partition_sequence_by_restricted_growth_function

sequencetools.**partition_sequence_by_restricted_growth_function**(*sequence*, *restricted_growth_function*)

New in version 2.0. Partition *sequence* by *restricted_growth_function*:

```
>>> l = range(10)
>>> rgf = [1, 1, 2, 2, 1, 2, 3, 3, 2, 4]
```

```
>>> sequencetools.partition_sequence_by_restricted_growth_function(l, rgf)
[[0, 1, 4], [2, 3, 5, 8], [6, 7], [9]]
```

Raise value error when *sequence* length does not equal *restricted_growth_function* length.

Return list of lists.

30.2.71 sequencetools.partition_sequence_by_sign_of_elements

`sequencetools.`**`partition_sequence_by_sign_of_elements`**(*sequence, sign=[-1, 0, 1]*)

New in version 1.1. Partition *sequence* elements by sign:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence))
[[0, 0], [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[-1]))
[0, 0, [-1, -1], 2, 3, [-5], 1, 2, 5, [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[0]))
[[0, 0], -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[1]))
[0, 0, -1, -1, [2, 3], -5, [1, 2, 5], -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[-1, 0]))
[[0, 0], [-1, -1], 2, 3, [-5], 1, 2, 5, [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[-1, 1]))
[0, 0, [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[0, 1]))
[[0, 0], -1, -1, [2, 3], -5, [1, 2, 5], -5, -6]
```

```
>>> list(sequencetools.partition_sequence_by_sign_of_elements(sequence, sign=[-1, 0, 1]))
[[0, 0], [-1, -1], [2, 3], [-5], [1, 2, 5], [-5, -6]]
```

When `-1` in `sign`, group negative elements.

When `0` in `sign`, group `0` elements.

When `1` in `sign`, group positive elements.

Return list of tuples of *sequence* element references.

30.2.72 sequencetools.partition_sequence_by_value_of_elements

`sequencetools.`**`partition_sequence_by_value_of_elements`**(*sequence*)

New in version 1.1. Group *sequence* elements by value of elements:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 1, 5, -5]
```

```
>>> sequencetools.partition_sequence_by_value_of_elements(sequence)
[(0, 0), (-1, -1), (2,), (3,), (-5,), (1, 1), (5,), (-5,)]
```

Return list of tuples of *sequence* element references.

30.2.73 sequencetools.partition_sequence_by_weights_at_least

sequencetools.**partition_sequence_by_weights_at_least**(*sequence*, *weights*, *cyclic=False*, *overhang=False*)

New in version 1.1. Partition *sequence* by *weights* at least.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5, 5]
```

Example 1. Partition sequence once by weights at least without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=False, overhang=False)
[[3, 3, 3, 3], [4]]
```

Example 2. Partition sequence once by weights at least with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=False, overhang=True)
[[3, 3, 3, 3], [4], [4, 4, 4, 5, 5]]
```

Example 3. Partition sequence cyclically by weights at least without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=True, overhang=False)
[[3, 3, 3, 3], [4], [4, 4, 4], [5]]
```

Example 4. Partition sequence cyclically by weights at least with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_least(
...     sequence, [10, 4], cyclic=True, overhang=True)
[[3, 3, 3, 3], [4], [4, 4, 4], [5], [5]]
```

Return list of sequence objects.

30.2.74 sequencetools.partition_sequence_by_weights_at_most

sequencetools.**partition_sequence_by_weights_at_most**(*sequence*, *weights*, *cyclic=False*, *overhang=False*)

New in version 1.1. Partition *sequence* by *weights* at most.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5, 5]
```

Example 1. Partition sequence once by weights at most without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence, [10, 4], cyclic=False, overhang=False)
[[3, 3, 3], [3]]
```

Example 2. Partition sequence once by weights at most with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence, [10, 4], cyclic=False, overhang=True)
[[3, 3, 3], [3], [4, 4, 4, 4, 5, 5]]
```

Example 3. Partition sequence cyclically by weights at most without overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence, [10, 5], cyclic=True, overhang=False)
[[3, 3, 3], [3], [4, 4], [4], [4, 5], [5]]
```

Example 4. Partition sequence cyclically by weights at most with overhang:

```
>>> sequencetools.partition_sequence_by_weights_at_most(
...     sequence, [10, 5], cyclic=True, overhang=True)
[[3, 3, 3], [3], [4, 4], [4], [4, 5], [5]]
```

Return list of sequence objects.

30.2.75 sequencetools.partition_sequence_by_weights_exactly

sequencetools.**partition_sequence_by_weights_exactly**(*sequence*, *weights*, *cyclic=False*, *overhang=False*)

New in version 1.1. Partition *sequence* by *weights* exactly.

```
>>> sequence = [3, 3, 3, 3, 4, 4, 4, 4, 5]
```

Example 1. Partition sequence once by weights exactly without overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence, [3, 9], cyclic=False, overhang=False)
[[3], [3, 3, 3]]
```

Example 2. Partition sequence once by weights exactly with overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence, [3, 9], cyclic=False, overhang=True)
[[3], [3, 3, 3], [4, 4, 4, 4, 5]]
```

Example 3. Partition sequence cyclically by weights exactly without overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence, [12], cyclic=True, overhang=False)
[[3, 3, 3, 3], [4, 4, 4]]
```

Example 4. Partition sequence cyclically by weights exactly with overhang:

```
>>> sequencetools.partition_sequence_by_weights_exactly(
...     sequence, [12], cyclic=True, overhang=True)
[[3, 3, 3, 3], [4, 4, 4], [4, 5]]
```

Return list sequence objects.

30.2.76 sequencetools.partition_sequence_extended_to_counts

sequencetools.**partition_sequence_extended_to_counts**(*sequence*, *counts*, *overhang=True*)

New in version 2.0. Partition sequence extended to counts.

Example 1. Partition sequence extended to counts with overhang:

```
>>> sequencetools.partition_sequence_extended_to_counts(
...     [1, 2, 3, 4], [6, 6, 6], overhang=True)
[[1, 2, 3, 4, 1, 2], [3, 4, 1, 2, 3, 4], [1, 2, 3, 4, 1, 2], [3, 4]]
```

Example 2. Partition sequence extended to coutns without overhang:

```
>>> sequencetools.partition_sequence_extended_to_counts(
...     [1, 2, 3, 4], [6, 6, 6], overhang=False)
[[1, 2, 3, 4, 1, 2], [3, 4, 1, 2, 3, 4], [1, 2, 3, 4, 1, 2]]
```

Return sequence of sequence objects.

30.2.77 sequencetools.permute_sequence

sequencetools.**permute_sequence**(*sequence*, *permutation*)

New in version 2.0. Permute *sequence* by *permutation*:

```
>>> sequencetools.permute_sequence([10, 11, 12, 13, 14, 15], [5, 4, 0, 1, 2, 3])
[15, 14, 10, 11, 12, 13]
```

Return newly constructed *sequence* object.

30.2.78 sequencetools.remove_sequence_elements_at_indices

sequencetools.**remove_sequence_elements_at_indices**(*sequence*, *indices*)

New in version 2.0. Remove *sequence* elements at *indices*:

```
>>> sequencetools.remove_sequence_elements_at_indices(range(20), [1, 16, 17, 18])
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 19]
```

Ignore negative indices.

Return list.

30.2.79 sequencetools.remove_sequence_elements_at_indices_cyclically

sequencetools.**remove_sequence_elements_at_indices_cyclically**(*sequence*, *indices*, *period*, *offset=0*)

New in version 2.0. Remove *sequence* elements at *indices* mod *period* plus *offset*:

```
>>> sequencetools.remove_sequence_elements_at_indices_cyclically(range(20), [0, 1], 5, 3)
[0, 1, 2, 5, 6, 7, 10, 11, 12, 15, 16, 17]
```

Ignore negative indices.

Return list.

30.2.80 sequencetools.remove_subsequence_of_weight_at_index

sequencetools.**remove_subsequence_of_weight_at_index**(*sequence*, *weight*, *index*)

New in version 1.1. Remove subsequence of *weight* at *index*:

```
>>> sequence = (1, 1, 2, 3, 5, 5, 1, 2, 5, 5, 6)
```

```
>>> sequencetools.remove_subsequence_of_weight_at_index(sequence, 13, 4)
(1, 1, 2, 3, 5, 5, 6)
```

Return newly constructed *sequence* object.

30.2.81 sequencetools.repeat_runs_in_sequence_to_count

sequencetools.**repeat_runs_in_sequence_to_count**(*sequence*, *indicators*)

New in version 1.1. Repeat subruns in *sequence* according to *indicators*. The *indicators* input parameter must be a list of zero or more `(start, length, count)` triples. For every `(start, length, count)` indicator in *indicators*, the function copies `sequence[start:start+length]` and inserts `count` new copies of `sequence[start:start+length]` immediately after `sequence[start:start+length]` in *sequence*.

The function reads the value of `count` in every `(start, length, count)` triple not as the total number of occurrences of `sequence[start:start+length]` to appear in *sequence* after execution, but rather as the number of new occurrences of `sequence[start:start+length]` to appear in *sequence* after execution.

The function wraps newly created subruns in tuples. That is, this function returns output with one more level of nesting than given in input.

To insert `10` count of `sequence[:2]` at `sequence[2:2]`:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(range(20), [(0, 2, 10)])
[0, 1, (0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1),
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

To insert `5` count of `sequence[10:12]` at `sequence[12:12]` and then insert `5` count of `sequence[:2]` at `sequence[2:2]`:

```
>>> sequence = range(20)
```

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(0, 2, 5), (10, 2, 5)])
[0, 1, (0, 1, 0, 1, 0, 1, 0, 1, 0, 1), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
(10, 11, 10, 11, 10, 11, 10, 11, 10, 11), 12, 13, 14, 15, 16, 17, 18, 19]
```

Note: This function wraps around the end of *sequence* whenever `len(sequence) < start + length`.

To insert `2` count of `[18, 19, 0, 1]` at `sequence[2:2]`:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(18, 4, 2)])
[0, 1, (18, 19, 0, 1, 18, 19, 0, 1), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19]
```

To insert `2` count of `[18, 19, 0, 1, 2, 3, 4]` at `sequence[4:4]`:

```
>>> sequencetools.repeat_runs_in_sequence_to_count(sequence, [(18, 8, 2)])
[0, 1, 2, 3, 4, 5, (18, 19, 0, 1, 2, 3, 4, 5, 18, 19, 0, 1, 2, 3, 4, 5), 6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Todo

Implement an optional *wrap* keyword to specify whether this function should wrap around the ened of *sequence* whenever `len(sequence) < start + length` or not.

Todo

Reimplement this function to return a generator.

Generalizations of this function would include functions to repeat subruns in *sequence* to not only a certain count, as implemented here, but to a certain length, weight or sum. That is, `sequencetools.repeat_subruns_to_length()`, `sequencetools.repeat_subruns_to_weight()` and `sequencetools.repeat_subruns_to_sum()`.

30.2.82 sequencetools.repeat_sequence_elements_at_indices

`sequencetools.`**`repeat_sequence_elements_at_indices`**(*sequence*, *indices*, *total*)

New in version 2.0. Repeat *sequence* elements at *indices* to *total* length:

```
>>> sequencetools.repeat_sequence_elements_at_indices(range(10), [6, 7, 8], 3)
[0, 1, 2, 3, 4, 5, [6, 6, 6], [7, 7, 7], [8, 8, 8], 9]
```

Return list.

30.2.83 sequencetools.repeat_sequence_elements_at_indices_cyclically

`sequencetools.`**`repeat_sequence_elements_at_indices_cyclically`**(*sequence*, *cycle_token*, *total*)

New in version 2.0. Repeat *sequence* elements at indices specified by *cycle_token* to *total* length:

```
>>> sequencetools.repeat_sequence_elements_at_indices_cyclically(range(10), (5, [1, 2]), 3)
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, [6, 6, 6], [7, 7, 7], 8, 9]
```

The *cycle_token* may be a sieve:

```
>>> sieve = sievetools.cycle_tokens_to_sieve((5, [1, 2]))
>>> sequencetools.repeat_sequence_elements_at_indices_cyclically(range(10), sieve, 3)
[0, [1, 1, 1], [2, 2, 2], 3, 4, 5, [6, 6, 6], [7, 7, 7], 8, 9]
```

Return list.

30.2.84 sequencetools.repeat_sequence_elements_n_times_each

sequencetools.**repeat_sequence_elements_n_times_each**(*sequence*, *n*)

New in version 1.1. Repeat *sequence* elements *n* times each:

```
>>> sequencetools.repeat_sequence_elements_n_times_each((1, -1, 2, -3, 5, -5, 6), 2)
(1, 1, -1, -1, 2, 2, -3, -3, 5, 5, -5, -5, 6, 6)
```

Return newly constructed *sequence* object with copied *sequence* elements.

30.2.85 sequencetools.repeat_sequence_n_times

sequencetools.**repeat_sequence_n_times**(*sequence*, *n*)

New in version 2.0. Repeat *sequence* *n* times:

```
>>> sequencetools.repeat_sequence_n_times((1, 2, 3, 4, 5), 3)
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

Repeat *sequence* `0` times:

```
>>> sequencetools.repeat_sequence_n_times((1, 2, 3, 4, 5), 0)
()
```

Return newly constructed *sequence* object of copied *sequence* elements.

30.2.86 sequencetools.repeat_sequence_to_length

sequencetools.**repeat_sequence_to_length**(*sequence*, *length*, *start=0*)

New in version 1.1. Repeat *sequence* to nonnegative integer *length*:

```
>>> sequencetools.repeat_sequence_to_length(range(5), 11)
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0]
```

Repeat *sequence* to nonnegative integer *length* from *start*:

```
>>> sequencetools.repeat_sequence_to_length(range(5), 11, start=2)
[2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2]
```

Return newly constructed *sequence* object.

30.2.87 sequencetools.repeat_sequence_to_weight_at_least

sequencetools.**repeat_sequence_to_weight_at_least**(*sequence*, *weight*)

New in version 1.1. Repeat *sequence* to *weight* at least:

```
>>> sequencetools.repeat_sequence_to_weight_at_least((5, -5, -5), 23)
(5, -5, -5, 5, -5)
```

Return newly constructed *sequence* object.

30.2.88 sequencetools.repeat_sequence_to_weight_at_most

sequencetools.**repeat_sequence_to_weight_at_most**(*sequence*, *weight*)
New in version 1.1. Repeat *sequence* to *weight* at most:

```
>>> sequencetools.repeat_sequence_to_weight_at_most((5, -5, -5), 23)
(5, -5, -5, 5)
```

Return newly constructed *sequence* object.

30.2.89 sequencetools.repeat_sequence_to_weight_exactly

sequencetools.**repeat_sequence_to_weight_exactly**(*sequence*, *weight*)
New in version 1.1. Repeat *sequence* to *weight* exactly:

```
>>> sequencetools.repeat_sequence_to_weight_exactly((5, -5, -5), 23)
(5, -5, -5, 5, -3)
```

Return newly constructed *sequence* object.

30.2.90 sequencetools.replace_sequence_elements_cyclically_with_new_material

sequencetools.**replace_sequence_elements_cyclically_with_new_material**(*sequence*, *indices*, *new_material*)
New in version 1.1. Replace *sequence* elements cyclically at *indices* with *new_material*:

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['A', 'B'], 3))
['A', 1, 'B', 3, 4, 5, 'A', 7, 'B', 9, 10, 11, 'A', 13, 'B', 15, 16, 17, 'A', 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['*'], 1))
['*', 1, '*', 3, '*', 5, '*', 7, '*', 9, '*', 11, '*', 13, '*', 15, '*', 17, '*', 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0], 2), (['A', 'B', 'C', 'D'], None))
['A', 1, 'B', 3, 'C', 5, 'D', 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

```
>>> sequencetools.replace_sequence_elements_cyclically_with_new_material(
... range(20), ([0, 1, 8, 13], None), (['A', 'B', 'C', 'D'], None))
['A', 'B', 2, 3, 4, 5, 6, 7, 'C', 9, 10, 11, 12, 'D', 14, 15, 16, 17, 18, 19]
```

Raise type error when *sequence* not a list.

Return new list.

30.2.91 sequencetools.retain_sequence_elements_at_indices

sequencetools.**retain_sequence_elements_at_indices**(*sequence*, *indices*)
New in version 2.0. Retain *sequence* elements at *indices*:

```
>>> sequencetools.retain_sequence_elements_at_indices(range(20), [1, 16, 17, 18])
[1, 16, 17, 18]
```

Return sequence elements in the order they appear in *sequence*.

Ignore negative indices.

Return list.

30.2.92 sequencetools.retain_sequence_elements_at_indices_cyclically

sequencetools.**retain_sequence_elements_at_indices_cyclically**(*sequence*, *indices*, *period*, *offset=0*)

New in version 2.0. Retain *sequence* elements at *indices* mod *period* plus *offset*:

```
>>> sequencetools.retain_sequence_elements_at_indices_cyclically(range(20), [0, 1], 5, 3)
[3, 4, 8, 9, 13, 14, 18, 19]
```

Ignore negative values in *indices*.

Return list.

30.2.93 sequencetools.reverse_sequence

sequencetools.**reverse_sequence**(*sequence*)

New in version 2.0. Reverse *sequence*:

```
>>> sequencetools.reverse_sequence((1, 2, 3, 4, 5))
(5, 4, 3, 2, 1)
```

Return new *sequence* object.

30.2.94 sequencetools.reverse_sequence_elements

sequencetools.**reverse_sequence_elements**(*sequence*)

New in version 2.0. Reverse *sequence* elements:

```
>>> sequencetools.reverse_sequence_elements([1, (2, 3, 4), 5, (6, 7)])
[1, (4, 3, 2), 5, (7, 6)]
```

Return new *sequence* object.

30.2.95 sequencetools.rotate_sequence

sequencetools.**rotate_sequence**(*sequence*, *n*)

New in version 1.1. Rotate *sequence* to the right:

```
>>> sequencetools.rotate_sequence(range(10), 4)
[6, 7, 8, 9, 0, 1, 2, 3, 4, 5]
```

Rotate *sequence* to the left:

```
>>> sequencetools.rotate_sequence(range(10), -3)
[3, 4, 5, 6, 7, 8, 9, 0, 1, 2]
```

Rotate *sequence* neither to the right nor the left:

```
>>> sequencetools.rotate_sequence(range(10), 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Return newly created *sequence* object.

30.2.96 sequencetools.splice_new_elements_between_sequence_elements

sequencetools.**splice_new_elements_between_sequence_elements**(*sequence*, *new_elements*, *overhang=(0, 0)*)

New in version 1.1. Splice copies of *new_elements* between each of the elements of *sequence*:

```
>>> sequence = [0, 1, 2, 3, 4]
>>> new_elements = ['A', 'B']
```

```
>>> sequencetools.splice_new_elements_between_sequence_elements(sequence, new_elements)
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splice copies of *new_elements* between each of the elements of *sequence* and after the last element of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(0, 1))
[0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4, 'A', 'B']
```

Splice copies of *new_elements* before the first element of *sequence* and between each of the other elements of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(1, 0))
['A', 'B', 0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4]
```

Splice copies of *new_elements* before the first element of *sequence*, after the last element of *sequence* and between each of the other elements of *sequence*:

```
>>> sequencetools.splice_new_elements_between_sequence_elements(
...     sequence, new_elements, overhang=(1, 1))
['A', 'B', 0, 'A', 'B', 1, 'A', 'B', 2, 'A', 'B', 3, 'A', 'B', 4, 'A', 'B']
```

Return newly constructed list.

30.2.97 sequencetools.split_sequence_by_weights

sequencetools.**split_sequence_by_weights**(*sequence*, *weights*, *cyclic=False*, *overhang=False*)

New in version 2.0. Split sequence by weights.

Example 1. Split sequence cyclically by weights with overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10), [3, 15, 3], cyclic=True, overhang=True)
[(3,), (7, -8), (-2, 1), (3,), (6, -9), (-1,)]
```

Example 2. Split sequence cyclically by weights without overhang:

```
>>> sequencetools.split_sequence_by_weights(
...         (10, -10, 10, -10), [3, 15, 3], cyclic=True, overhang=False)
[(3,), (7, -8), (-2, 1), (3,), (6, -9)]
```

Example 3. Split sequence once by weights with overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10), [3, 15, 3], cyclic=False, overhang=True)
[(3,), (7, -8), (-2, 1), (9, -10)]
```

Example 4. Split sequence once by weights without overhang:

```
>>> sequencetools.split_sequence_by_weights(
...     (10, -10, 10, -10), [3, 15, 3], cyclic=False, overhang=False)
[(3,), (7, -8), (-2, 1)]
```

Return list of sequence types.

30.2.98 sequencetools.split_sequence_extended_to_weights

sequencetools.**split_sequence_extended_to_weights**(*sequence*, *weights*, *overhang=True*)

New in version 2.0. Split sequence extended to weights.

Example 1. Split sequence extended to weights with overhang:

```
>>> sequencetools.split_sequence_extended_to_weights(
...     [1, 2, 3, 4, 5], [7, 7, 7], overhang=True)
[[1, 2, 3, 1], [3, 4], [1, 1, 2, 3], [4, 5]]
```

Example 2. Split sequence extended to weights without overhang:

```
>>> sequencetools.split_sequence_extended_to_weights(
...     [1, 2, 3, 4, 5], [7, 7, 7], overhang=False)
[[1, 2, 3, 1], [3, 4], [1, 1, 2, 3]]
```

Return sequence of sequence objects.

30.2.99 sequencetools.sum_consecutive_sequence_elements_by_sign

sequencetools.**sum_consecutive_sequence_elements_by_sign**(*sequence, sign=[-1, 0, 1]*)

New in version 1.1. Sum consecutive *sequence* elements by *sign*:

```
>>> sequence = [0, 0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence)
[0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1])
[0, 0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[0])
[0, -1, -1, 2, 3, -5, 1, 2, 5, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[1])
[0, 0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 0])
[0, -2, 2, 3, -5, 1, 2, 5, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 1])
[0, 0, -2, 5, -5, 8, -11]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[0, 1])
[0, -1, -1, 5, -5, 8, -5, -6]
```

```
>>> sequencetools.sum_consecutive_sequence_elements_by_sign(sequence, sign=[-1, 0, 1])
[0, -2, 5, -5, 8, -11]
```

When `-1` in *sign*, sum consecutive negative elements.

When `0` in *sign*, sum consecutive `0` elements.

When `1` in *sign*, sum consecutive positive elements.

Return list.

30.2.100 sequencetools.sum_sequence_elements_at_indices

sequencetools.**sum_sequence_elements_at_indices**(*sequence, pairs, period=None, overhang=True*)

New in version 1.1. Sum *sequence* elements at indices according to *pairs*:

```
>>> sequencetools.sum_sequence_elements_at_indices(range(10), [(0, 3)])
[3, 3, 4, 5, 6, 7, 8, 9]
```

Sum *sequence* elements cyclically at indices according to *pairs* and *period*:

```
>>> sequencetools.sum_sequence_elements_at_indices(range(10), [(0, 3)], period=4)
[3, 3, 15, 7, 17]
```

Sum *sequence* elements cyclically at indices according to *pairs* and *period* and do not return incomplete final sum:

```
>>> sequencetools.sum_sequence_elements_at_indices(
...     range(10), [(0, 3)], period=4, overhang=False)
[3, 3, 15, 7]
```

Replace `sequence[i:i+count]` with `sum(sequence[i:i+count])` for each `(i, count)` in *pairs*.

Indices in *pairs* must be less than *period* when *period* is not none.

Return new list.

30.2.101 sequencetools.truncate_runs_in_sequence

`sequencetools.`**`truncate_runs_in_sequence`**`(`*sequence*`)`

New in version 1.1. Truncate subruns of like elements in *sequence* to length `1`:

```
>>> sequencetools.truncate_runs_in_sequence([1, 1, 2, 3, 3, 3, 9, 4, 4, 4])
[1, 2, 3, 9, 4]
```

Return empty list when *sequence* is empty:

```
>>> sequencetools.truncate_runs_in_sequence([])
[]
```

Raise type error when *sequence* is not a list.

Return new list.

30.2.102 sequencetools.truncate_sequence_to_sum

`sequencetools.`**`truncate_sequence_to_sum`**`(`*sequence*, *target_sum*`)`

New in version 1.1. Truncate *sequence* to *target_sum*:

```
>>> sequence = [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

```
>>> for n in range(10):
...     print n, sequencetools.truncate_sequence_to_sum(sequence, n)
...
0 []
1 [-1, 2]
2 [-1, 2, -3, 4]
3 [-1, 2, -3, 4, -5, 6]
4 [-1, 2, -3, 4, -5, 6, -7, 8]
5 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
6 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
7 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
8 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
9 [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
```

Return empty list when *target_sum* is `0`:

```
>>> sequencetools.truncate_sequence_to_sum([1, 2, 3, 4, 5], 0)
[]
```

Raise type error when *sequence* is not a list.

Raise value error on negative *target_sum*.

Return new list.

30.2.103 sequencetools.truncate_sequence_to_weight

sequencetools.**truncate_sequence_to_weight**(*sequence*, *weight*)

New in version 1.1. Truncate *sequence* to *weight*:

```
>>> l = [-1, 2, -3, 4, -5, 6, -7, 8, -9, 10]
>>> for x in range(10):
...     print x, sequencetools.truncate_sequence_to_weight(l, x)
...
0 []
1 [-1]
2 [-1, 1]
3 [-1, 2]
4 [-1, 2, -1]
5 [-1, 2, -2]
6 [-1, 2, -3]
7 [-1, 2, -3, 1]
8 [-1, 2, -3, 2]
9 [-1, 2, -3, 3]
```

Return empty list when *weight* is `0`:

```
>>> sequencetools.truncate_sequence_to_weight([1, 2, 3, 4, 5], 0)
[]
```

Raise type error when *sequence* is not a list.

Raise value error on negative *weight*.

Return new list.

30.2.104 sequencetools.yield_all_combinations_of_sequence_elements

sequencetools.**yield_all_combinations_of_sequence_elements**(*sequence*, *min_length=None*, *max_length=None*)

New in version 2.0. Yield all combinations of *sequence* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3], [4], [1, 4],
[2, 4], [1, 2, 4], [3, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

Yield all combinations of *sequence* greater than or equal to *min_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], min_length=3))
[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

Yield all combinations of *sequence* less than or equal to *max_length* in binary string order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], max_length=2))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [4], [1, 4], [2, 4], [3, 4]]
```

Yield all combiantions of *sequence* greater than or equal to *min_length* and less than or equal to *max_length* in lex order:

```
>>> list(sequencetools.yield_all_combinations_of_sequence_elements(
...     [1, 2, 3, 4], min_length=2, max_length=2))
[[1, 2], [1, 3], [2, 3], [1, 4], [2, 4], [3, 4]]
```

Return generator of newly created *sequence* objects.

30.2.105 sequencetools.yield_all_k_ary_sequences_of_length

sequencetools.**yield_all_k_ary_sequences_of_length**(*k*, *length*)
New in version 2.0. Generate all *k*-ary sequences of *length*:

```
>>> for sequence in sequencetools.yield_all_k_ary_sequences_of_length(2, 3):
...     sequence
...
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
```

Return generator of tuples.

30.2.106 sequencetools.yield_all_pairs_between_sequences

sequencetools.**yield_all_pairs_between_sequences**(*l*, *m*)
New in version 2.0. Yield all pairs between sequences *l* and *m*:

```
>>> for pair in sequencetools.yield_all_pairs_between_sequences([1, 2, 3], [4, 5]):
...     pair
...
(1, 4)
(1, 5)
(2, 4)
(2, 5)
(3, 4)
(3, 5)
```

Return pair generator.

30.2.107 sequencetools.yield_all_partitions_of_sequence

sequencetools.**yield_all_partitions_of_sequence**(*sequence*)
New in version 2.0. Yield all partitions of *sequence*:

```
>>> for partition in sequencetools.yield_all_partitions_of_sequence([0, 1, 2, 3]):
...     partition
...
[[0, 1, 2, 3]]
[[0, 1, 2], [3]]
[[0, 1], [2, 3]]
[[0, 1], [2], [3]]
[[0], [1, 2, 3]]
[[0], [1, 2], [3]]
[[0], [1], [2, 3]]
[[0], [1], [2], [3]]
```

Return generator of newly created lists.

30.2.108 sequencetools.yield_all_permutations_of_sequence

sequencetools.**yield_all_permutations_of_sequence**(*sequence*)
New in version 1.1. Yield all permutations of *sequence* in lex order:

```
>>> list(sequencetools.yield_all_permutations_of_sequence((1, 2, 3)))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

Return generator of *sequence* objects.

30.2.109 sequencetools.yield_all_permutations_of_sequence_in_orbit

sequencetools.**yield_all_permutations_of_sequence_in_orbit**(*sequence*, *permutation*)

New in version 2.0. Yield all permutations of *sequence* in orbit of *permutation* in lex order:

```
>>> list(sequencetools.yield_all_permutations_of_sequence_in_orbit(
...     (1, 2, 3, 4), [1, 2, 3, 0]))
[(1, 2, 3, 4), (2, 3, 4, 1), (3, 4, 1, 2), (4, 1, 2, 3)]
```

Return generator of *sequence* objects.

30.2.110 sequencetools.yield_all_restricted_growth_functions_of_length

sequencetools.**yield_all_restricted_growth_functions_of_length**(*length*)

New in version 2.0. Generate all restricted growth functions of *length* in lex order:

```
>>> for rgf in sequencetools.yield_all_restricted_growth_functions_of_length(4):
...     rgf
...
(1, 1, 1, 1)
(1, 1, 1, 2)
(1, 1, 2, 1)
(1, 1, 2, 2)
(1, 1, 2, 3)
(1, 2, 1, 1)
(1, 2, 1, 2)
(1, 2, 1, 3)
(1, 2, 2, 1)
(1, 2, 2, 2)
(1, 2, 2, 3)
(1, 2, 3, 1)
(1, 2, 3, 2)
(1, 2, 3, 3)
(1, 2, 3, 4)
```

Return generator of tuples.

30.2.111 sequencetools.yield_all_rotations_of_sequence

sequencetools.**yield_all_rotations_of_sequence**(*sequence*, *n=1*)

New in version 2.0. Yield all *n*-rotations of *sequence* up to identity:

```
>>> list(sequencetools.yield_all_rotations_of_sequence([1, 2, 3, 4], -1))
[[1, 2, 3, 4], [2, 3, 4, 1], [3, 4, 1, 2], [4, 1, 2, 3]]
```

Return generator of *sequence* objects.

30.2.112 sequencetools.yield_all_set_partitions_of_sequence

sequencetools.**yield_all_set_partitions_of_sequence**(*sequence*)

New in version 2.0. Yield all set partitions of *sequence* in restricted growth function order:

```
>>> for set_partition in sequencetools.yield_all_set_partitions_of_sequence(
...     [21, 22, 23, 24]):
...     set_partition
...
[[21, 22, 23, 24]]
[[21, 22, 23], [24]]
[[21, 22, 24], [23]]
[[21, 22], [23, 24]]
[[21, 22], [23], [24]]
[[21, 23, 24], [22]]
[[21, 23], [22, 24]]
```

```
[[21, 23], [22], [24]]
[[21, 24], [22, 23]]
[[21], [22, 23, 24]]
[[21], [22, 23], [24]]
[[21, 24], [22], [23]]
[[21], [22, 24], [23]]
[[21], [22], [23, 24]]
[[21], [22], [23], [24]]
```

Return generator of list of lists.

30.2.113 sequencetools.yield_all_subsequences_of_sequence

sequencetools.**yield_all_subsequences_of_sequence**(*sequence*, *min_length=0*, *max_length=None*)

New in version 2.0. Yield all subsequences of *sequence* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence([0, 1, 2]))
[[], [0], [0, 1], [0, 1, 2], [1], [1, 2], [2]]
```

Yield all subsequences of *sequence* greater than or equal to *min_length* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], min_length=3))
[[0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [1, 2, 3], [1, 2, 3, 4], [2, 3, 4]]
```

Yield all subsequences of *sequence* less than or equal to *max_length* in lex order:

```
>>> for subsequence in sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], max_length=3):
...     subsequence
[]
[0]
[0, 1]
[0, 1, 2]
[1]
[1, 2]
[1, 2, 3]
[2]
[2, 3]
[2, 3, 4]
[3]
[3, 4]
[4]
```

Yield all subsequences of *sequence* greater than or equal to *min_length* and less than or equal to *max_length* in lex order:

```
>>> list(sequencetools.yield_all_subsequences_of_sequence(
...     [0, 1, 2, 3, 4], min_length=3, max_length=3))
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

Return generator of newly created *sequence* slices.

30.2.114 sequencetools.yield_all_unordered_pairs_of_sequence

sequencetools.**yield_all_unordered_pairs_of_sequence**(*sequence*)

New in version 2.0. Yield all unordered pairs of *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1, 2, 3, 4]))
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

Yield all unordered pairs of length-`1` *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1]))
[]
```

Yield all unordered pairs of empty *sequence*:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([]))
[]
```

Yield all unordered pairs of *sequence* with duplicate elements:

```
>>> list(sequencetools.yield_all_unordered_pairs_of_sequence([1, 1, 1]))
[(1, 1), (1, 1), (1, 1)]
```

Pairs are tuples instead of sets to accommodate duplicate *sequence* elements.

Return generator.

30.2.115 sequencetools.yield_outer_product_of_sequences

sequencetools.**yield_outer_product_of_sequences**(*sequences*)

New in version 1.1. Yield outer product of *sequences*:

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], ['a', 'b']]))
[[1, 'a'], [1, 'b'], [2, 'a'], [2, 'b'], [3, 'a'], [3, 'b']]
```

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], ['a', 'b'], ['X', 'Y']]))
[[1, 'a', 'X'], [1, 'a', 'Y'], [1, 'b', 'X'], [1, 'b', 'Y'],
[2, 'a', 'X'], [2, 'a', 'Y'], [2, 'b', 'X'], [2, 'b', 'Y'],
[3, 'a', 'X'], [3, 'a', 'Y'], [3, 'b', 'X'], [3, 'b', 'Y']]
```

```
>>> list(sequencetools.yield_outer_product_of_sequences(
...     [[1, 2, 3], [4, 5], [6, 7, 8]]))
[[1, 4, 6], [1, 4, 7], [1, 4, 8], [1, 5, 6], [1, 5, 7], [1, 5, 8],
[2, 4, 6], [2, 4, 7], [2, 4, 8], [2, 5, 6], [2, 5, 7], [2, 5, 8],
[3, 4, 6], [3, 4, 7], [3, 4, 8], [3, 5, 6], [3, 5, 7], [3, 5, 8]]
```

Return generator.

30.2.116 sequencetools.zip_sequences_cyclically

sequencetools.**zip_sequences_cyclically**(**sequences*)

New in version 1.1. Zip *sequences* cyclically:

```
>>> sequencetools.zip_sequences_cyclically([1, 2, 3], ['a', 'b'])
[(1, 'a'), (2, 'b'), (3, 'a')]
```

New in version 1.1: Arbitrary number of input sequences now allowed.

```
>>> sequencetools.zip_sequences_cyclically([10, 11, 12], [20, 21], [30, 31, 32, 33])
[(10, 20, 30), (11, 21, 31), (12, 20, 32), (10, 21, 33)]
```

Cycle over the elements of the sequences of shorter length.

Return list of length equal to sequence of greatest length in *sequences*.

30.2.117 sequencetools.zip_sequences_without_truncation

sequencetools.**zip_sequences_without_truncation**(**sequences*)

New in version 1.1. Zip *sequences* nontruncating:

```
>>> sequencetools.zip_sequences_without_truncation(
...     [1, 2, 3, 4], [11, 12, 13], [21, 22, 23])
[(1, 11, 21), (2, 12, 22), (3, 13, 23), (4,)]
```

Lengths of the tuples returned may differ but will always be greater than or equal to `1`.

Return list of tuples.

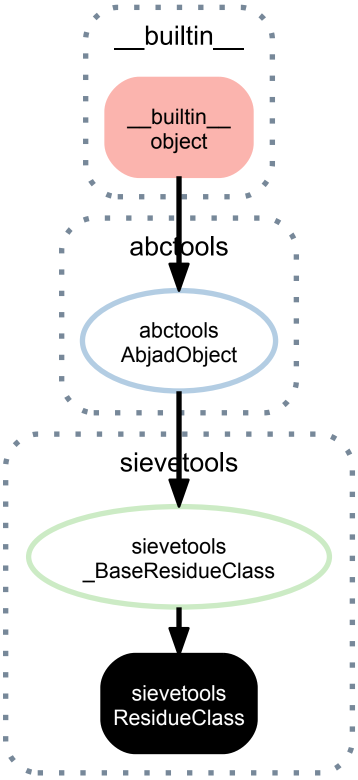
CHAPTER

THIRTYONE

SIEVETOOLS

31.1 Concrete Classes

31.1.1 sievetools.ResidueClass



class sievetools.**ResidueClass**(**args*)

Residue class (or congruence class).

Residue classes form the basis of Xenakis sieves. They can be used to make any complex periodic integer or boolean sequence as a combination of simple periodic sequences.

Example from the opening of Xenakis's *Psappha* for solo percussion:

```
>>> from abjad.tools.sievetools import ResidueClass as RC
```

```
>>> s1 = (RC(8, 0) | RC(8, 1) | RC(8, 7)) & (RC(5, 1) | RC(5, 3))
>>> s2 = (RC(8, 0) | RC(8, 1) | RC(8, 2)) & RC(5, 0)
>>> s3 = RC(8, 3)
>>> s4 = RC(8, 4)
>>> s5 = (RC(8, 5) | RC(8, 6)) & (RC(5, 2) | RC(5, 3) | RC(5, 4))
```

```
>>> s6 = (RC(8, 1) & RC(5, 2))
>>> s7 = (RC(8, 6) & RC(5, 1))
```

```
>>> y = s1 | s2 | s3 | s4 | s5 | s6 | s7
```

```
>>> y.get_congruent_bases(40)
    [0, 1, 3, 4, 6, 8, 10, 11, 12, 13, 14, 16, 17, 19, 20, 22, 23, 25, 27,
    28, 29, 31, 33, 35, 36, 37, 38, 40]
```

```
>>> y.get_boolean_train(40)
    [1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
    1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0]
```

Return residue class.

Read-only properties

`ResidueClass.`**`modulo`**
Period of residue class.

`ResidueClass.`**`residue`**
Residue of residue class.

`ResidueClass.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`ResidueClass.`**`get_boolean_train`**`(`*`*min_max`*`)`
Returns a boolean train with 0s mapped to the integers that are not congruent bases of the residue class and 1s mapped to those that are. The method takes one or two integer arguments. If only one is given, it is taken as the max range and the min is assumed to be 0.

Example:

```
>>> r = RC(3, 0)
>>> r.get_boolean_train(6)
[1, 0, 0, 1, 0, 0]
>>> r.get_congruent_bases(-6, 6)
[-6, -3, 0, 3, 6]
```

Return list.

`ResidueClass.`**`get_congruent_bases`**`(`*`*min_max`*`)`
Returns all the congruent bases of this residue class within the given range. The method takes one or two integer arguments. If only one it given, it is taken as the max range and the min is assumed to be 0.

Example:

```
>>> r = RC(3, 0)
>>> r.get_congruent_bases(6)
[0, 3, 6]
>>> r.get_congruent_bases(-6, 6)
[-6, -3, 0, 3, 6]
```

Return list.

Special methods

`ResidueClass.`**`__and__`**`(`*`arg`*`)`

`ResidueClass.`**`__eq__`**`(`*`exp`*`)`

ResidueClass.**__ge__**(*expr*)

ResidueClass.**__gt__**(*expr*)

ResidueClass.**__le__**(*expr*)

ResidueClass.**__lt__**(*expr*)

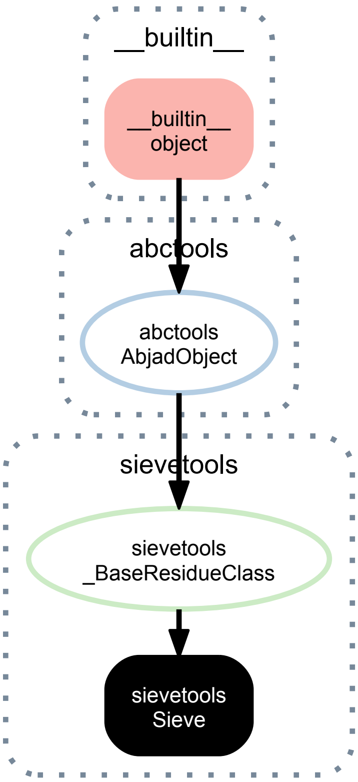
ResidueClass.**__ne__**(*expr*)

ResidueClass.**__or__**(*arg*)

ResidueClass.**__repr__**()

ResidueClass.**__xor__**(*arg*)

31.1.2 sievetools.Sieve



class sievetools.**Sieve**(*rcs*, *logical_operator='or'*)

Read-only properties

Sieve.**logical_operator**
Residue class expression logical operator.

Sieve.**period**
Residue class expression period.

Sieve.**rcs**
Residue class expression residue classes.

Sieve.**representative_boolean_train**
Residue class expression representative boolean train.

Sieve.**representative_congruent_bases**
Residue class expression representative congruent bases.

`Sieve.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`Sieve.`**`get_boolean_train`**`(`**min_max*`)`
Returns a boolean train with 0s mapped to the integers that are not congruent bases of the RC expression and 1s mapped to those that are. The method takes one or two integer arguments. If only one is given, it is taken as the max range and min is assumed to be 0.

Example:

```
>>> from abjad.tools.sievetools import ResidueClass as RC
```

```
>>> e = RC(3, 0) | RC(2, 0)
>>> e.get_boolean_train(6)
[1, 0, 1, 1, 1, 0]
>>> e.get_congruent_bases(-6, 6)
[-6, -4, -3, -2, 0, 2, 3, 4, 6]
```

Return list.

`Sieve.`**`get_congruent_bases`**`(`**min_max*`)`
Returns all the congruent bases of this RC expression within the given range. The method takes one or two integer arguments. If only one it given, it is taken as the max range and min is assumed to be 0.

Example:

```
>>> e = RC(3, 0) | RC(2, 0)
>>> e.get_congruent_bases(6)
[0, 2, 3, 4, 6]
>>> e.get_congruent_bases(-6, 6)
[-6, -4, -3, -2, 0, 2, 3, 4, 6]
```

Return list.

`Sieve.`**`is_congruent_base`**`(`*integer*`)`

Special methods

`Sieve.`**`__and__`**`(`*arg*`)`

`Sieve.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Sieve.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Sieve.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Sieve.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Sieve.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

Sieve.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Sieve.**__or__**(*arg*)

Sieve.**__repr__**()

Sieve.**__xor__**(*arg*)

31.2 Functions

31.2.1 sievetools.all_are_residue_class_expressions

sievetools.**all_are_residue_class_expressions**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad residue class expressions:

```
>>> sieve = sievetools.ResidueClass(3, 0) | sievetools.ResidueClass(2, 0)
```

```
>>> sieve
{ResidueClass(2, 0) | ResidueClass(3, 0)}
```

```
>>> sievetools.all_are_residue_class_expressions([sieve])
True
```

True when *expr* is an empty sequence:

```
>>> sievetools.all_are_residue_class_expressions([])
True
```

Otherwise false:

```
>>> sievetools.all_are_residue_class_expressions('foo')
False
```

Return boolean.

31.2.2 sievetools.cycle_tokens_to_sieve

sievetools.**cycle_tokens_to_sieve**(**cycle_tokens*)
New in version 2.0. Make Xenakis sieve from arbitrarily many *cycle_tokens*:

```
>>> cycle_token_1 = (6, [0, 4, 5])
>>> cycle_token_2 = (10, [0, 1, 2], 6)
>>> sievetools.cycle_tokens_to_sieve(cycle_token_1, cycle_token_2)
{ResidueClass(6, 0) | ResidueClass(6, 4) | ResidueClass(6, 5) |
 ResidueClass(10, 6) | ResidueClass(10, 7) | ResidueClass(10, 8)}
```

Cycle token comprises *modulo*, *residues* and optional *offset*.

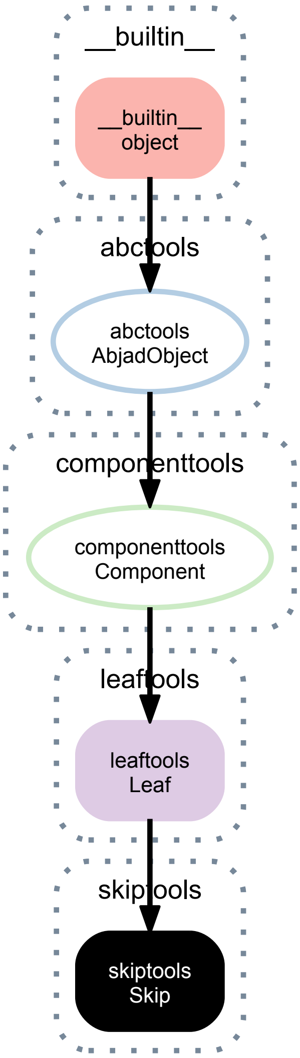
CHAPTER

THIRTYTWO

SKIPTOOLS

32.1 Concrete Classes

32.1.1 skiptools.Skip



class skiptools.**Skip**(**args*, ***kwargs*)

Abjad model of a LilyPond skip:

```
>>> skiptools.Skip((3, 16))
Skip('s8.')
```

Return Skip instance.

Read-only properties

`Skip.`**`descendants`**
Read-only reference to component descendants score selection.

`Skip.`**`duration`**

`Skip.`**`duration_in_seconds`**

`Skip.`**`leaf_index`**

`Skip.`**`lilypond_format`**

`Skip.`**`lineage`**
Read-only reference to component lineage score selection.

`Skip.`**`multiplied_duration`**

`Skip.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Skip.`**`parent`**

`Skip.`**`parentage`**
Read-only reference to component parentage score selection.

`Skip.`**`preprolated_duration`**

`Skip.`**`prolation`**

`Skip.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Skip.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`Skip.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Skip.`**`timespan`**
Read-only timespan of component.

`Skip.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Read/write properties

`Skip.`**`duration_multiplier`**

`Skip.`**`written_duration`**

`Skip.`**`written_pitch_indication_is_at_sounding_pitch`**

`Skip.`**`written_pitch_indication_is_nonsemantic`**

Special methods

`Skip.`**`__and__`**(*arg*)

`Skip.`**`__copy__`**(**args*)

Skip.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Skip.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Skip.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Skip.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Skip.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Skip.**__mul__**(*n*)

Skip.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Skip.**__or__**(*arg*)

Skip.**__repr__**()

Skip.**__rmul__**(*n*)

Skip.**__str__**()

Skip.**__sub__**(*arg*)

Skip.**__xor__**(*arg*)

32.2 Functions

32.2.1 skiptools.all_are_skips

skiptools.**all_are_skips**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad skips:

```
>>> skips = 3 * skiptools.Skip('s4')
```

```
>>> skips
[Skip('s4'), Skip('s4'), Skip('s4')]
```

```
>>> skiptools.all_are_skips(skips)
True
```

True when *expr* is an empty sequence:

```
>>> skiptools.all_are_skips([])
True
```

Otherwise false:

```
>>> skiptools.all_are_skips('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

32.2.2 skiptools.make_repeated_skips_from_time_signature

skiptools.**make_repeated_skips_from_time_signature**(*time_signature*)

New in version 2.0. Make repeated skips from *time_signature*:

```
>>> skiptools.make_repeated_skips_from_time_signature((5, 32))
[Skip('s32'), Skip('s32'), Skip('s32'), Skip('s32'), Skip('s32')]
```

Return list of skips.

32.2.3 skiptools.make_repeated_skips_from_time_signatures

skiptools.**make_repeated_skips_from_time_signatures**(*time_signatures*)

Make repated skips from *time_signatures*:

```
skiptools.make_repeated_skips_from_time_signatures([(2, 8), (3, 32)])
[[Skip('s8'), Skip('s8')], [Skip('s32'), Skip('s32'), Skip('s32')]]
```

Return list of skip lists.

32.2.4 skiptools.make_skips_with_multiplied_durations

skiptools.**make_skips_with_multiplied_durations**(*written_duration*, *multiplied_durations*)

New in version 2.0. Make *written_duration* skips with *multiplied_durations*:

```
>>> skiptools.make_skips_with_multiplied_durations(
...     Duration(1, 4), [(1, 2), (1, 3), (1, 4), (1, 5)])
[Skip('s4 * 2'), Skip('s4 * 4/3'), Skip('s4 * 1'), Skip('s4 * 4/5')]
```

Useful for making invisible layout voices.

Return list of skips.

32.2.5 skiptools.replace_leaves_in_expr_with_skips

skiptools.**replace_leaves_in_expr_with_skips**(*expr*)

New in version 1.1. Replace leaves in *expr* with skips:

```
>>> staff = Staff(Measure((2, 8), "c'8 d'8") * 2)
>>> skiptools.replace_leaves_in_expr_with_skips(staff[0])
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        s8
        s8
    }
    {
        c'8
        d'8
    }
}
```

Return none.

32.2.6 skiptools.yield_groups_of_skips_in_sequence

skiptools.**yield_groups_of_skips_in_sequence**(*sequence*)
New in version 2.0. Yield groups of skips in *sequence*:

```
>>> staff = Staff("c'8 d'8 s8 s8 <e' g'>8 <f' a'>8 g'8 a'8 s8 s8 <b' d''>8 <c'' e''>8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    s8
    s8
    <e' g'>8
    <f' a'>8
    g'8
    a'8
    s8
    s8
    <b' d''>8
    <c'' e''>8
}
```

```
>>> for skip in skiptools.yield_groups_of_skips_in_sequence(staff):
...     skip
...
(Skip('s8'), Skip('s8'))
(Skip('s8'), Skip('s8'))
```

Return generator.

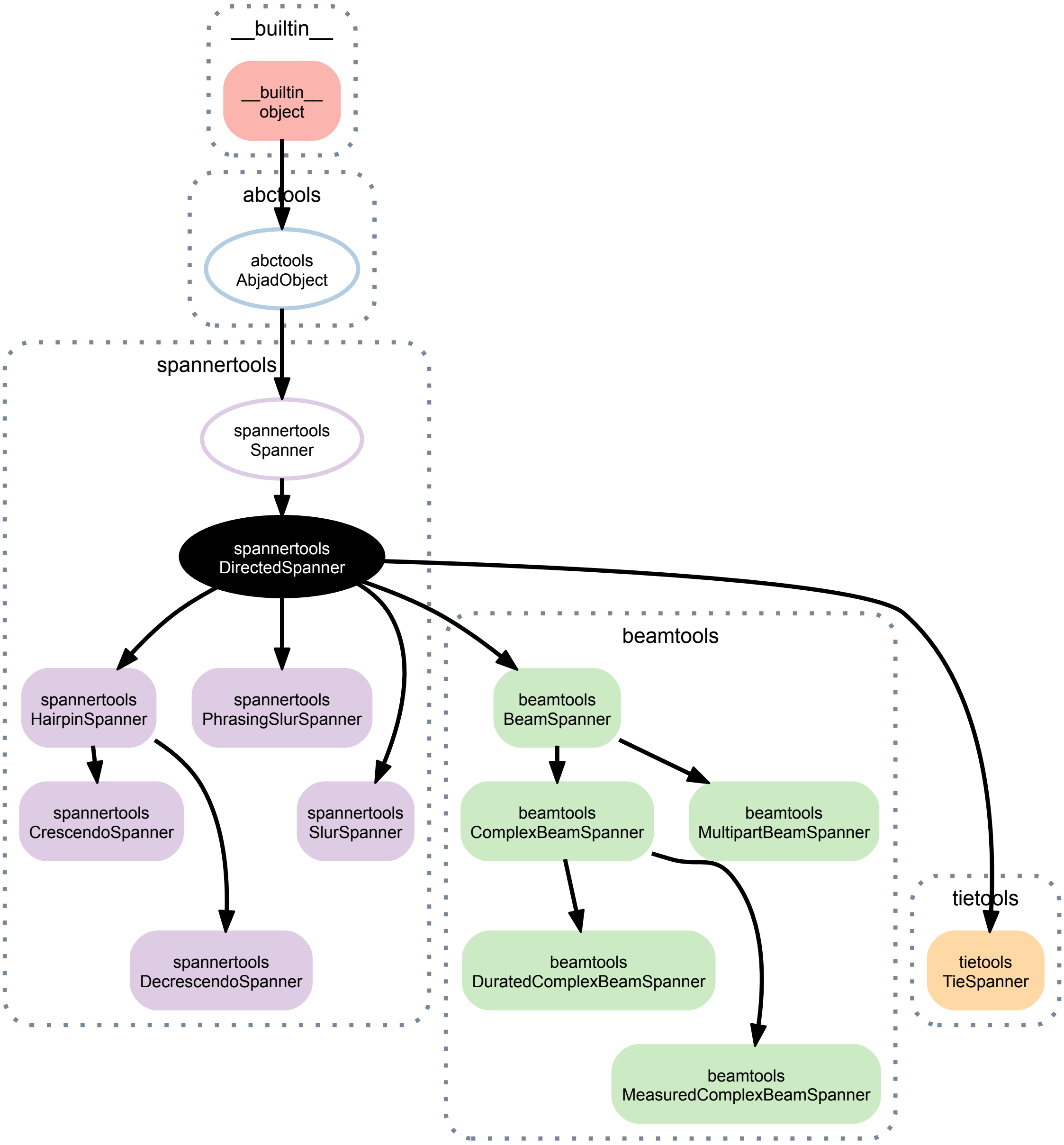
CHAPTER

THIRTYTHREE

SPANNERTOOLS

33.1 Abstract Classes

33.1.1 spannertools.DirectedSpanner



class spannertools.**DirectedSpanner**(*components=[ ]*, *direction=None*)
Abstract Spanner subclass for spanners which may take an "up" or "down" indication.

Read-only properties

DirectedSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

DirectedSpanner.**duration**
Sum of prolated duration of all components in spanner.

DirectedSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

DirectedSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

DirectedSpanner.**override**
LilyPond grob override component plug-in.

DirectedSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

DirectedSpanner.**set**
LilyPond context setting component plug-in.

DirectedSpanner.**storage_format**
Storage format of Abjad object.

Return string.

DirectedSpanner.**timespan**
Read-only timespan of spanner.

DirectedSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

DirectedSpanner.**direction**

Methods

DirectedSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`DirectedSpanner.`**`append_left`**`(`*component*`)`
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`DirectedSpanner.`**`clear`**`()`
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`DirectedSpanner.`**`extend`**`(`*components*`)`
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DirectedSpanner.`**`extend_left`**`(`*components*`)`
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DirectedSpanner.`**`fracture`**`(`*i*, *direction=None*`)`
Fracture spanner at *direction* of component at index *i*.

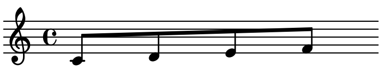
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

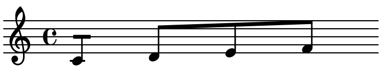
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`DirectedSpanner.`**`fuse`**`(`*spanner*`)`
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`DirectedSpanner.`**`index`**`(`*component*`)`
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`DirectedSpanner.pop()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

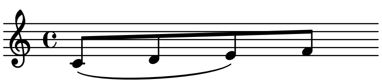
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

`DirectedSpanner.pop_left()`

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

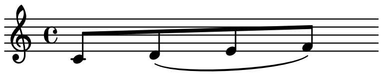
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

DirectedSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

DirectedSpanner.**__contains__**(*expr*)

DirectedSpanner.**__copy__**(**args*)

DirectedSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DirectedSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectedSpanner.**__getitem__**(*expr*)

DirectedSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DirectedSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectedSpanner.**__len__**()

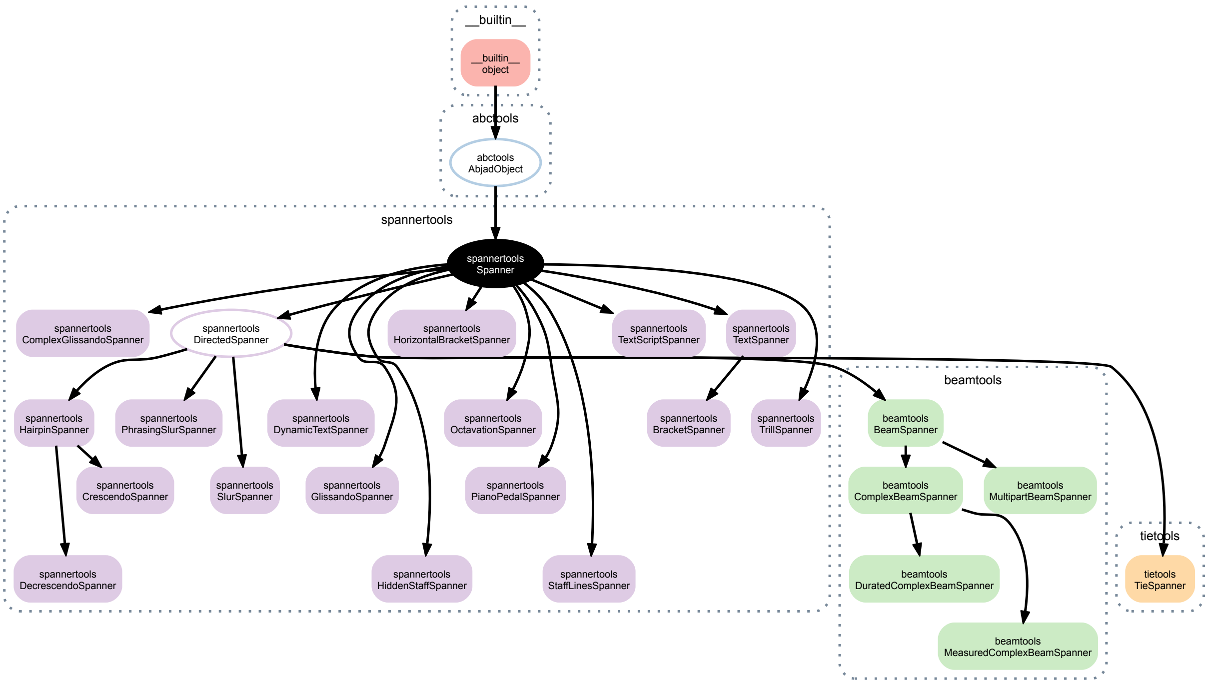
DirectedSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

DirectedSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DirectedSpanner.**__repr__**()

33.1.2 spannertools.Spanner



class spannertools.**Spanner**(*components=None*)
New in version 1.1. Any type of notation object that stretches horizontally and encompasses some number of notes, rest, chords, tuplets, measures, voices or other Abjad components.

Beams, slurs, hairpins, trills, glissandi and piano pedal brackets all stretch horizontally on the page to encompass multiple notes and all implement as Abjad spanners. That is, these spanner all have an obvious graphic reality with definite start-, stop- and midpoints.

Abjad also implements a number of spanners of a different type, such as tempo and instrument spanners, which mark a group of notes, rests, chords or measues as carrying a certain tempo or being played by a certain instrument.

The spanner class described here abstracts the functionality that all such spanners, both graphic and non-graphics, share. This shared functionality includes methods to add, remove, inspect and test components governed by the spanner, as well as basic formatting properties. The other spanner classes, such as beam and glissando, all inherit from this class and receive the functionality implemented here.

Read-only properties

Spanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

Spanner.**duration**
Sum of prolated duration of all components in spanner.

Spanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

Spanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

Spanner.**override**
LilyPond grob override component plug-in.

Spanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

Spanner.**set**
LilyPond context setting component plug-in.

Spanner.**storage_format**
Storage format of Abjad object.

Return string.

Spanner.**timespan**
Read-only timespan of spanner.

Spanner.**written_duration**
Sum of written duration of all components in spanner.

Methods

Spanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

Spanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`Spanner.clear()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`Spanner.extend(`*components*`)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`Spanner.extend_left(`*components*`)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`Spanner.fracture(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

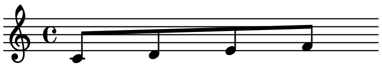
Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
```

```
    f'8 ]
}
```

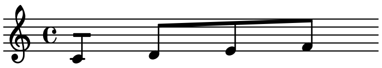
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`Spanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`Spanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

Spanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

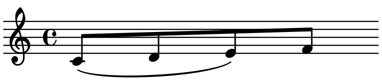
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

Spanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

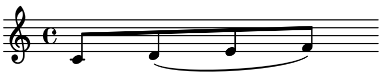
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

Spanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

Spanner.**__contains__**(*expr*)

Spanner.**__copy__**(**args*)

Spanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Spanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Spanner.**__getitem__**(*expr*)

Spanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Spanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Spanner.**__len__**()

Spanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

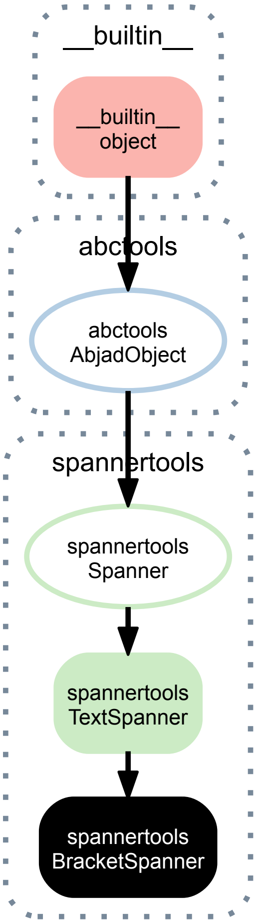
Spanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Spanner.**__repr__**()

33.2 Concrete Classes

33.2.1 spannertools.BracketSpanner



class spannertools.**BracketSpanner**(*components=None*)
Abjad bracket spanner:

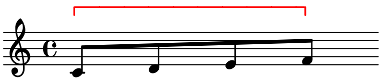
```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.BracketSpanner(staff[:])
BracketSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    \override TextSpanner #'bound-details #'left #'text = \markup {
        \draw-line #'(0 . -1) }
    \override TextSpanner #'bound-details #'left-broken #'text = ##f
    \override TextSpanner #'bound-details #'right #'text = \markup {
```

```
            \draw-line #'(0 . -1) }
        \override TextSpanner #'bound-details #'right-broken #'text = ##f
        \override TextSpanner #'color = #red
        \override TextSpanner #'dash-fraction = #1
        \override TextSpanner #'staff-padding = #2
        \override TextSpanner #'thickness = #1.5
        c'8 \startTextSpan
        d'8
        e'8
        f'8 \stopTextSpan
        \revert TextSpanner #'bound-details #'left #'text
        \revert TextSpanner #'bound-details #'left-broken #'text
        \revert TextSpanner #'bound-details #'right #'text
        \revert TextSpanner #'bound-details #'right-broken #'text
        \revert TextSpanner #'color
        \revert TextSpanner #'dash-fraction
        \revert TextSpanner #'staff-padding
        \revert TextSpanner #'thickness
    }
```

```
>>> show(staff)
```



Render 1.5-unit thick solid red spanner.

Draw nibs at beginning and end of spanner.

Do not draw nibs at line breaks.

Return bracket spanner.

Read-only properties

BracketSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

BracketSpanner.**duration**

Sum of prolated duration of all components in spanner.

BracketSpanner.**duration_in_seconds**

Sum of duration of all leaves in spanner, in seconds.

BracketSpanner.**leaves**

Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

BracketSpanner.**override**

LilyPond grob override component plug-in.

BracketSpanner.**preprolated_duration**

Sum of preprolated duration of all components in spanner.

BracketSpanner.**set**
LilyPond context setting component plug-in.

BracketSpanner.**storage_format**
Storage format of Abjad object.

Return string.

BracketSpanner.**timespan**
Read-only timespan of spanner.

BracketSpanner.**written_duration**
Sum of written duration of all components in spanner.

Methods

BracketSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

BracketSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

BracketSpanner.**clear**()
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

BracketSpanner.**extend**(*components*)
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

BracketSpanner.**extend_left**(*components*)
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

BracketSpanner.**fracture**(*i*, *direction=None*)
Fracture spanner at *direction* of component at index *i*.

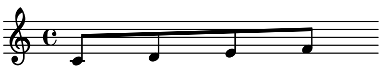
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

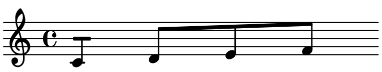
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

BracketSpanner.**fuse**(*spanner*)
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`BracketSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`BracketSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

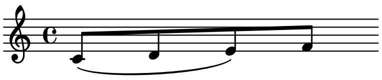
```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d'8
        e'8 )
        f'8
    }
```

```
>>> show(voice)
```



Return component.

BracketSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

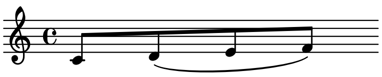
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

BracketSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

`BracketSpanner.`**`__contains__`**`(`*expr*`)`

`BracketSpanner.`**`__copy__`**`(`**args*`)`

`BracketSpanner.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`BracketSpanner.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`BracketSpanner.`**`__getitem__`**`(`*expr*`)`

`BracketSpanner.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`BracketSpanner.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`BracketSpanner.`**`__len__`**`()`

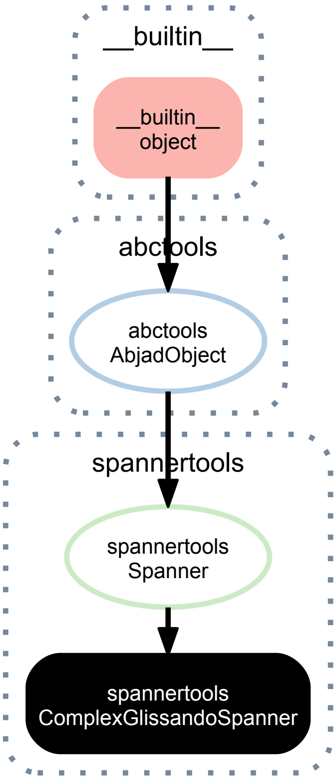
`BracketSpanner.`**`__lt__`**`(`*expr*`)`
Trivial comparison to allow doctests to work.

`BracketSpanner.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`BracketSpanner.`**`__repr__`**`()`

33.2.2 spannertools.ComplexGlissandoSpanner



class spannertools.**ComplexGlissandoSpanner**(*components=None*)

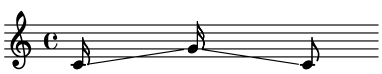
New in version 2.9. Abjad rest-skipping glissando spanner:

```
>>> staff = Staff("c'16 r r g' r8 c'8")
```

```
>>> spannertools.ComplexGlissandoSpanner(staff[:])
ComplexGlissandoSpanner(c'16, r16, r16, g'16, r8, c'8)
```

```
>>> f(staff)
\new Staff {
    c'16 \glissando
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    g'16 \glissando
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r8
    c'8
}
```

```
>>> show(staff)
```



Should be used with beamtools.BeamSpanner for best effect, along with an override of Stem #'stemlet-length, in order to generate stemlets over each invisible rest.

Format nonlast leaves in spanner with LilyPond glissando command.

Set all Rest instances to transparent.

Set all NoteColumns filled with silences to be skipped by glissandi.

Return *ComplexGlissandoSpanner* instance.

Read-only properties

`ComplexGlissandoSpanner.`**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`ComplexGlissandoSpanner.`**duration**

Sum of prolated duration of all components in spanner.

`ComplexGlissandoSpanner.`**duration_in_seconds**

Sum of duration of all leaves in spanner, in seconds.

`ComplexGlissandoSpanner.`**leaves**

Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`ComplexGlissandoSpanner.`**override**

LilyPond grob override component plug-in.

`ComplexGlissandoSpanner.`**preprolated_duration**

Sum of preprolated duration of all components in spanner.

`ComplexGlissandoSpanner.`**set**

LilyPond context setting component plug-in.

`ComplexGlissandoSpanner.`**storage_format**

Storage format of Abjad object.

Return string.

`ComplexGlissandoSpanner.`**timespan**

Read-only timespan of spanner.

`ComplexGlissandoSpanner.`**written_duration**

Sum of written duration of all components in spanner.

Methods

`ComplexGlissandoSpanner.`**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

ComplexGlissandoSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

ComplexGlissandoSpanner.**clear**()
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

ComplexGlissandoSpanner.**extend**(*components*)
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

ComplexGlissandoSpanner.**extend_left**(*components*)
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

ComplexGlissandoSpanner.**fracture**(*i*, *direction=None*)
Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

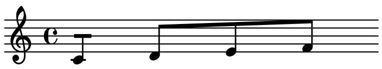
```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

ComplexGlissandoSpanner.**fuse**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

ComplexGlissandoSpanner.**index**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`ComplexGlissandoSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

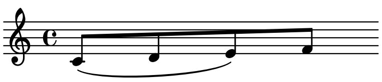
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

`ComplexGlissandoSpanner.`**`pop_left`**`()`

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

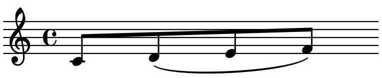
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

ComplexGlissandoSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

ComplexGlissandoSpanner.**__contains__**(*expr*)

ComplexGlissandoSpanner.**__copy__**(**args*)

ComplexGlissandoSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ComplexGlissandoSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ComplexGlissandoSpanner.**__getitem__**(*expr*)

ComplexGlissandoSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ComplexGlissandoSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ComplexGlissandoSpanner.__len__()`

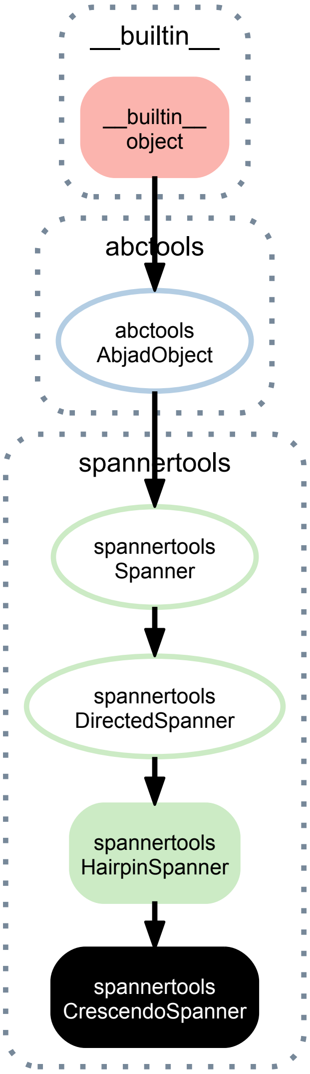
`ComplexGlissandoSpanner.__lt__(expr)`
Trivial comparison to allow doctests to work.

`ComplexGlissandoSpanner.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`ComplexGlissandoSpanner.__repr__()`

33.2.3 spannertools.CrescendoSpanner



class `spannertools.CrescendoSpanner(components=None, include_rests=True, direction=None)`
Abjad crescendo spanner that includes rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> f(staff)
\new Staff {
    r4
    c'8
    d'8
    e'8
    f'8
```

```
        r4
    }
```

```
    >>> spannertools.CrescendoSpanner(staff[:], include_rests=True)
    CrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
    >>> f(staff)
    \new Staff {
        r4 \<
        c'8
        d'8
        e'8
        f'8
        r4 \!
    }
```

```
    >>> show(staff)
```

Abjad crescendo spanner that does not include rests:

```
    >>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
    >>> f(staff)
    \new Staff {
        r4
        c'8
        d'8
        e'8
        f'8
        r4
    }
```

```
    >>> spannertools.CrescendoSpanner(staff[:], include_rests=False)
    CrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
    >>> f(staff)
    \new Staff {
        r4
        c'8 \<
        d'8
        e'8
        f'8 \!
        r4
    }
```

```
    >>> show(staff)
```

Return crescendo spanner.

Read-only properties

`CrescendoSpanner.`**`components`**

Return read-only tuple of components in spanner:

```
    >>> voice = Voice("c'8 d'8 e'8 f'8")
    >>> spanner = beamtools.BeamSpanner(voice[:2])
    >>> spanner.components
    (Note("c'8"), Note("d'8"))
```

Return tuple.

CrescendoSpanner.**duration**
Sum of prolated duration of all components in spanner.

CrescendoSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

CrescendoSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

CrescendoSpanner.**override**
LilyPond grob override component plug-in.

CrescendoSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

CrescendoSpanner.**set**
LilyPond context setting component plug-in.

CrescendoSpanner.**storage_format**
Storage format of Abjad object.

Return string.

CrescendoSpanner.**timespan**
Read-only timespan of spanner.

CrescendoSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

CrescendoSpanner.**direction**

CrescendoSpanner.**include_rests**
Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

CrescendoSpanner.**shape_string**
Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Set string.

CrescendoSpanner.**start_dynamic_string**
Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

CrescendoSpanner.**stop_dynamic_string**
Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

Methods

CrescendoSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

CrescendoSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`CrescendoSpanner.clear()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`CrescendoSpanner.extend(components)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`CrescendoSpanner.extend_left(components)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`CrescendoSpanner.fracture(i, direction=None)`

Fracture spanner at *direction* of component at index *i*.

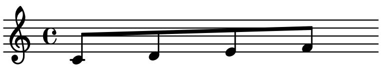
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

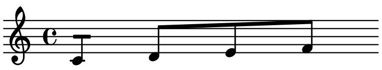
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

CrescendoSpanner.**fuse**(*spanner*)
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

CrescendoSpanner.**index**(*component*)
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

CrescendoSpanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

CrescendoSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

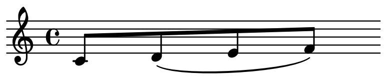
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

CrescendoSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

CrescendoSpanner.**__contains__**(*expr*)

CrescendoSpanner.**__copy__**(**args*)

CrescendoSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

CrescendoSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CrescendoSpanner.**__getitem__**(*expr*)

CrescendoSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

CrescendoSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CrescendoSpanner.**__len__**()

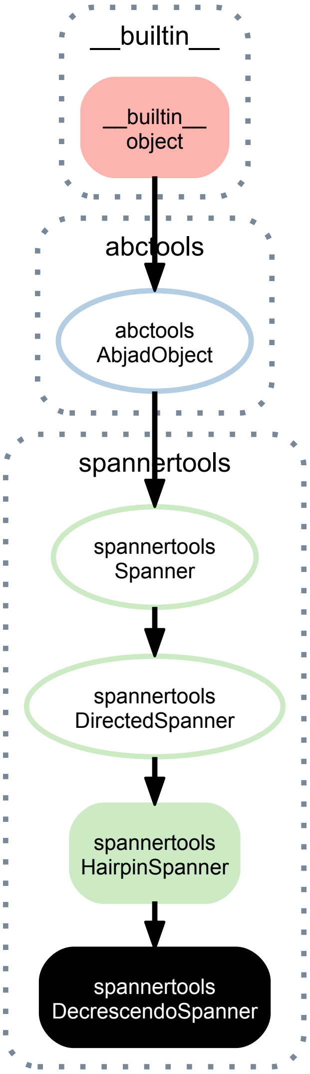
`CrescendoSpanner.__lt__(expr)`
Trivial comparison to allow doctests to work.

`CrescendoSpanner.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`CrescendoSpanner.__repr__()`

33.2.4 spannertools.DecrescendoSpanner



class `spannertools.DecrescendoSpanner`(*components=None*, *include_rests=True*, *direction=None*)
Abjad decrescendo spanner that includes rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> f(staff)
\new Staff {
    r4
    c'8
    d'8
    e'8
    f'8
    r4
}
```

```
>>> spannertools.DecrescendoSpanner(staff[:], include_rests=True)
DecrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> f(staff)
\new Staff {
    r4 \>
    c'8
    d'8
    e'8
    f'8
    r4 \!
}
```

```
>>> show(staff)
```

Abjad decrescendo spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
```

```
>>> f(staff)
\new Staff {
    r4
    c'8
    d'8
    e'8
    f'8
    r4
}
```

```
>>> spannertools.DecrescendoSpanner(staff[:], include_rests=False)
DecrescendoSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> f(staff)
\new Staff {
    r4
    c'8 \>
    d'8
    e'8
    f'8 \!
    r4
}
```

```
>>> show(staff)
```

Return decrescendo spanner.

Read-only properties

`DecrescendoSpanner.`**`components`**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`DecrescendoSpanner.`**`duration`**

Sum of prolated duration of all components in spanner.

DecrescendoSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

DecrescendoSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

DecrescendoSpanner.**override**
LilyPond grob override component plug-in.

DecrescendoSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

DecrescendoSpanner.**set**
LilyPond context setting component plug-in.

DecrescendoSpanner.**storage_format**
Storage format of Abjad object.

Return string.

DecrescendoSpanner.**timespan**
Read-only timespan of spanner.

DecrescendoSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

DecrescendoSpanner.**direction**

DecrescendoSpanner.**include_rests**
Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

DecrescendoSpanner.**shape_string**
Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string = '>'
```

```
>>> hairpin.shape_string
'>'
```

Set string.

DecrescendoSpanner.**start_dynamic_string**

Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

DecrescendoSpanner.**stop_dynamic_string**

Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

Methods

DecrescendoSpanner.**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

DecrescendoSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`DecrescendoSpanner.clear()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`DecrescendoSpanner.extend(components)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DecrescendoSpanner.extend_left(components)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`DecrescendoSpanner.fracture(i, direction=None)`

Fracture spanner at *direction* of component at index *i*.

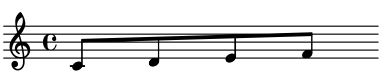
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

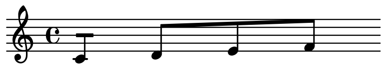
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`DecrescendoSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`DecrescendoSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`DecrescendoSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

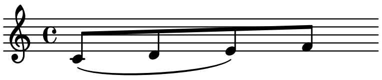
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

DecrescendoSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

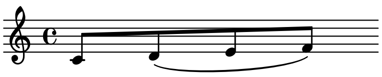
```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
```

```
        f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

DecrescendoSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

DecrescendoSpanner.**__contains__**(*expr*)

DecrescendoSpanner.**__copy__**(**args*)

DecrescendoSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DecrescendoSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DecrescendoSpanner.**__getitem__**(*expr*)

DecrescendoSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DecrescendoSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DecrescendoSpanner.**__len__**()

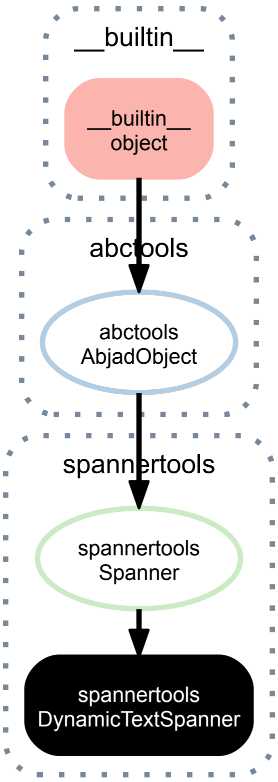
DecrescendoSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

DecrescendoSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DecrescendoSpanner.**__repr__**()

33.2.5 spannertools.DynamicTextSpanner



class spannertools.**DynamicTextSpanner**(*components=None*, *mark=''*)

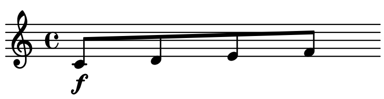
Abjad dynamic text spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.DynamicTextSpanner(staff[:], 'f')
DynamicTextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \f
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



Format dynamic *mark* at first leaf in spanner.

Return dynamic text spanner.

Read-only properties

DynamicTextSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

DynamicTextSpanner.**duration**
Sum of prolated duration of all components in spanner.

DynamicTextSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

DynamicTextSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

DynamicTextSpanner.**override**
LilyPond grob override component plug-in.

DynamicTextSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

DynamicTextSpanner.**set**
LilyPond context setting component plug-in.

DynamicTextSpanner.**storage_format**
Storage format of Abjad object.

Return string.

DynamicTextSpanner.**timespan**
Read-only timespan of spanner.

DynamicTextSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

DynamicTextSpanner.**mark**
Get dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> dynamic_text_spanner = spannertools.DynamicTextSpanner(staff[:], 'f')
>>> dynamic_text_spanner.mark
'f'
```

Set dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> dynamic_text_spanner = spannertools.DynamicTextSpanner(staff[:], 'f')
>>> dynamic_text_spanner.mark = 'p'
>>> dynamic_text_spanner.mark
'p'
```

Set string.

Methods

DynamicTextSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

DynamicTextSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

DynamicTextSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

DynamicTextSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

DynamicTextSpanner.**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

DynamicTextSpanner.**fracture**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

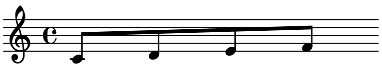
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

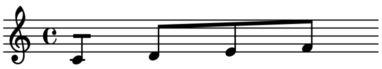
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`DynamicTextSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

DynamicTextSpanner.**index**(*component*)
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

DynamicTextSpanner.**pop**()
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

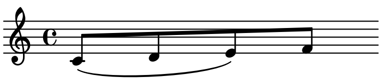
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

DynamicTextSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d’8
        e’8
        f’8 )
}
```

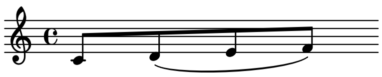
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c’8")
```

```
>>> spanner
SlurSpanner(d’8, e’8, f’8)
```

```
>>> f(voice)
\new Voice {
    c’8
    d’8 (
    e’8
    f’8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

DynamicTextSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c’8 d’8 e’8 f’8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c’8, d’8, e’8, f’8)
```

```
>>> f(staff)
\new Staff {
    c’8 [
    d’8
    e’8
    f’8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

DynamicTextSpanner.**__contains__**(*expr*)

DynamicTextSpanner.**__copy__**(**args*)

DynamicTextSpanner.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

DynamicTextSpanner.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

DynamicTextSpanner.**__getitem__**(*expr*)

DynamicTextSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DynamicTextSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DynamicTextSpanner.**__len__**()

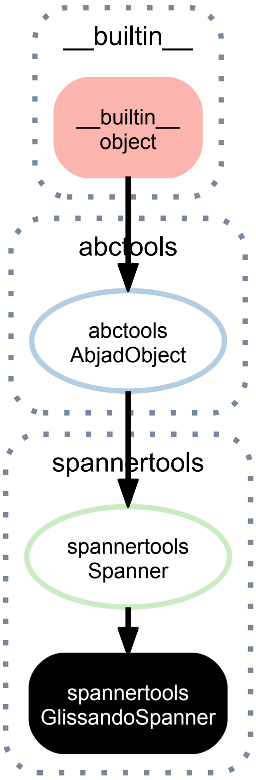
DynamicTextSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

DynamicTextSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DynamicTextSpanner.**__repr__**()

33.2.6 spannertools.GlissandoSpanner



class spannertools.**GlissandoSpanner**(*components=None*)
Abjad glissando spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.GlissandoSpanner(staff[:])
GlissandoSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \glissando
    d'8 \glissando
```

```
        e'8 \glissando
        f'8
    }
```

```
>>> show(staff)
```

Format nonlast leaves in spanner with LilyPond glissando command.

Return glissando spanner.

Read-only properties

`GlissandoSpanner.components`
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`GlissandoSpanner.duration`
Sum of prolated duration of all components in spanner.

`GlissandoSpanner.duration_in_seconds`
Sum of duration of all leaves in spanner, in seconds.

`GlissandoSpanner.leaves`
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`GlissandoSpanner.override`
LilyPond grob override component plug-in.

`GlissandoSpanner.preprolated_duration`
Sum of preprolated duration of all components in spanner.

`GlissandoSpanner.set`
LilyPond context setting component plug-in.

`GlissandoSpanner.storage_format`
Storage format of Abjad object.

Return string.

`GlissandoSpanner.timespan`
Read-only timespan of spanner.

`GlissandoSpanner.written_duration`
Sum of written duration of all components in spanner.

Methods

`GlissandoSpanner.append(component)`
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

GlissandoSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

GlissandoSpanner.**clear**()
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

GlissandoSpanner.**extend**(*components*)
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

GlissandoSpanner.**extend_left**(*components*)
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

GlissandoSpanner.**fracture**(*i*, *direction=None*)
Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

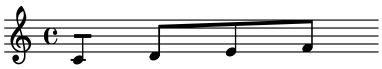
```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`GlissandoSpanner.`**`fuse`**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

GlissandoSpanner.**index**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

GlissandoSpanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

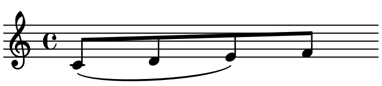
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

GlissandoSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d'8
        e'8
        f'8 )
}
```

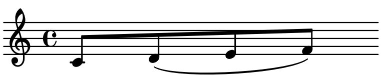
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

GlissandoSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

GlissandoSpanner.**__contains__**(*expr*)

GlissandoSpanner.**__copy__**(**args*)

GlissandoSpanner.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

GlissandoSpanner.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

GlissandoSpanner.**__getitem__**(*expr*)

GlissandoSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GlissandoSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GlissandoSpanner.**__len__**()

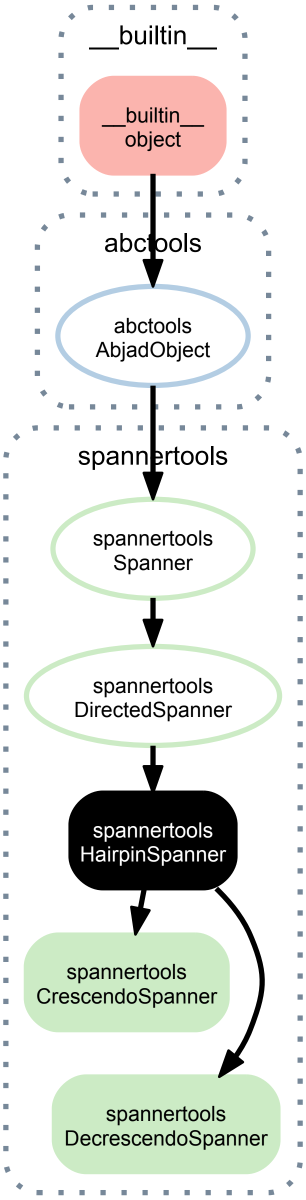
GlissandoSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

GlissandoSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

GlissandoSpanner.**__repr__**()

33.2.7 spannertools.HairpinSpanner



class `spannertools.HairpinSpanner`(*components=None*, *descriptor='<'*, *include_rests=True*, *direction=None*)

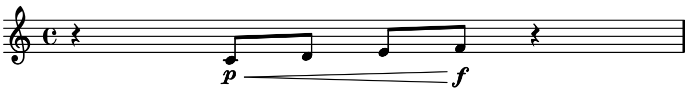
Hairpin spanner.

Example 1. Hairpin spanner that does not include rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=False)
HairpinSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> f(staff)
\new Staff {
    r4
    c'8 \< \p
    d'8
    e'8
    f'8 \f
    r4
}
```

```
>>> show(staff)
```



Example 2. Hairpin spanner that includes rests:

```
>>> staff = Staff("r4 c'8 d'8 e'8 f'8 r4")
>>> spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
HairpinSpanner(r4, c'8, d'8, e'8, f'8, r4)
```

```
>>> f(staff)
\new Staff {
    r4 \< \p
    c'8
    d'8
    e'8
    f'8
    r4 \f
}
```

```
>>> show(staff)
```

Return hairpin spanner.

Read-only properties

HairpinSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

HairpinSpanner.**duration**
Sum of prolated duration of all components in spanner.

HairpinSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

HairpinSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

HairpinSpanner.**override**
LilyPond grob override component plug-in.

HairpinSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

HairpinSpanner.**set**
LilyPond context setting component plug-in.

HairpinSpanner.**storage_format**

Storage format of Abjad object.

Return string.

HairpinSpanner.**timespan**

Read-only timespan of spanner.

HairpinSpanner.**written_duration**

Sum of written duration of all components in spanner.

Read/write properties

HairpinSpanner.**direction**

HairpinSpanner.**include_rests**

Get boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests
True
```

Set boolean hairpin rests setting:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f', include_rests=True)
>>> hairpin.include_rests = False
>>> hairpin.include_rests
False
```

Set boolean.

HairpinSpanner.**shape_string**

Get hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string
'<'
```

Set hairpin shape string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.shape_string = '>'
>>> hairpin.shape_string
'>'
```

Set string.

HairpinSpanner.**start_dynamic_string**

Get hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string
'p'
```

Set hairpin start dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.start_dynamic_string = 'mf'
>>> hairpin.start_dynamic_string
'mf'
```

Set string.

HairpinSpanner.**stop_dynamic_string**

Get hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string
'f'
```

Set hairpin stop dynamic string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> hairpin = spannertools.HairpinSpanner(staff[:], 'p < f')
>>> hairpin.stop_dynamic_string = 'mf'
>>> hairpin.stop_dynamic_string
'mf'
```

Set string.

Methods

HairpinSpanner.**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

HairpinSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

HairpinSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

HairpinSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`HairpinSpanner.`**`extend_left`**`(`*components*`)`
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`HairpinSpanner.`**`fracture`**`(`*i*, *direction=None*`)`
Fracture spanner at *direction* of component at index *i*.

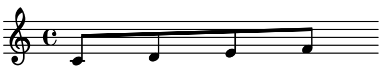
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

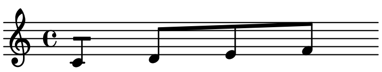
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`HairpinSpanner.`**`fuse`**`(`*spanner*`)`
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`HairpinSpanner.index(component)`
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`HairpinSpanner.pop()`
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d'8
        e'8 )
        f'8
}
```

```
>>> show(voice)
```

Return component.

HairpinSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

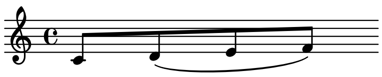
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

HairpinSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

HairpinSpanner.**__contains__**(*expr*)

HairpinSpanner.**__copy__**(**args*)

HairpinSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

HairpinSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HairpinSpanner.**__getitem__**(*expr*)

HairpinSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HairpinSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HairpinSpanner.**__len__**()

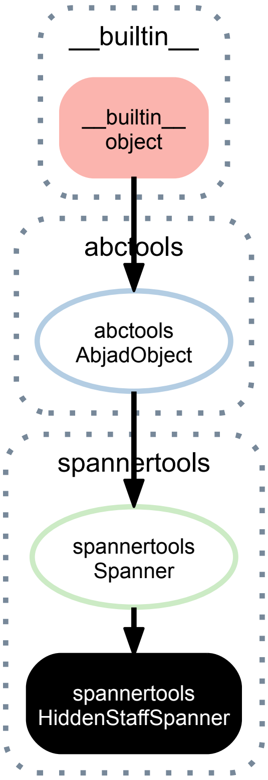
HairpinSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

HairpinSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

HairpinSpanner.**__repr__**()

33.2.8 spannertools.HiddenStaffSpanner



class `spannertools.HiddenStaffSpanner` (*components=None*)

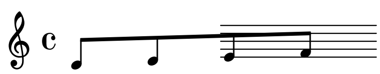
Abjad hidden staff spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.HiddenStaffSpanner(staff[:2])
HiddenStaffSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    \stopStaff
    c'8
    d'8
    \startStaff
    e'8
    f'8
}
```

```
>>> show(staff)
```



Hide staff behind leaves in spanner.

Return hidden staff spanner.

Read-only properties

`HiddenStaffSpanner.components`

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
```

```
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

HiddenStaffSpanner.**duration**
Sum of prolated duration of all components in spanner.

HiddenStaffSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

HiddenStaffSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

HiddenStaffSpanner.**override**
LilyPond grob override component plug-in.

HiddenStaffSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

HiddenStaffSpanner.**set**
LilyPond context setting component plug-in.

HiddenStaffSpanner.**storage_format**
Storage format of Abjad object.

Return string.

HiddenStaffSpanner.**timespan**
Read-only timespan of spanner.

HiddenStaffSpanner.**written_duration**
Sum of written duration of all components in spanner.

Methods

HiddenStaffSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

HiddenStaffSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

HiddenStaffSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

HiddenStaffSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

HiddenStaffSpanner.**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

HiddenStaffSpanner.**fracture**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

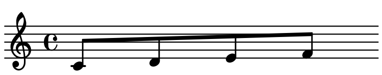
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

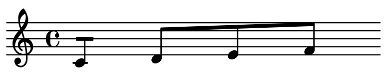
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`HiddenStaffSpanner.`**`fuse`**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`HiddenStaffSpanner.`**`index`**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`HiddenStaffSpanner.`**`pop`**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

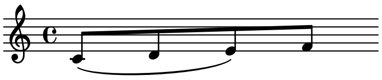
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

HiddenStaffSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

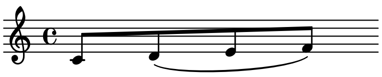
```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
```

```
        f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

HiddenStaffSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

HiddenStaffSpanner.**__contains__**(*expr*)

HiddenStaffSpanner.**__copy__**(**args*)

HiddenStaffSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

HiddenStaffSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HiddenStaffSpanner.**__getitem__**(*expr*)

HiddenStaffSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HiddenStaffSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HiddenStaffSpanner.**__len__**()

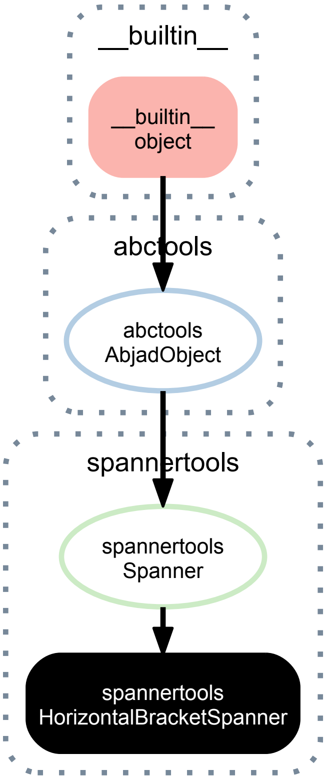
HiddenStaffSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

HiddenStaffSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

HiddenStaffSpanner.**__repr__**()

33.2.9 spannertools.HorizontalBracketSpanner



class spannertools.**HorizontalBracketSpanner**(*components=None*)

New in version 2.4. Abjad horizontal bracket spanner:

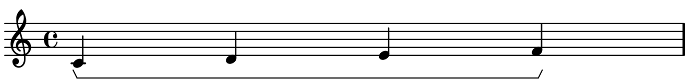
```
>>> voice = Voice("c'4 d'4 e'4 f'4")
>>> voice.engraver_consists.append('Horizontal_bracket_engraver')
```

```
>>> horizontal_bracket_spanner = spannertools.HorizontalBracketSpanner(voice[:])
```

```
>>> horizontal_bracket_spanner
HorizontalBracketSpanner(c'4, d'4, e'4, f'4)
```

```
>>> f(voice)
\new Voice \with {
    \consists Horizontal_bracket_engraver
} {
    c'4 \startGroup
    d'4
    e'4
    f'4 \stopGroup
}
```

```
>>> show(voice)
```



Return horizontal bracket spanner.

Read-only properties

HorizontalBracketSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

HorizontalBracketSpanner.**duration**
Sum of prolated duration of all components in spanner.

HorizontalBracketSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

HorizontalBracketSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

HorizontalBracketSpanner.**override**
LilyPond grob override component plug-in.

HorizontalBracketSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

HorizontalBracketSpanner.**set**
LilyPond context setting component plug-in.

HorizontalBracketSpanner.**storage_format**
Storage format of Abjad object.

Return string.

HorizontalBracketSpanner.**timespan**
Read-only timespan of spanner.

HorizontalBracketSpanner.**written_duration**
Sum of written duration of all components in spanner.

Methods

HorizontalBracketSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

HorizontalBracketSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`HorizontalBracketSpanner.`**`clear`**`()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`HorizontalBracketSpanner.`**`extend`**`(`*components*`)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`HorizontalBracketSpanner.`**`extend_left`**`(`*components*`)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`HorizontalBracketSpanner.`**`fracture`**`(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

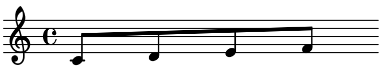
Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
```

```
    f'8 ]
}
```

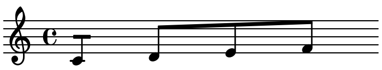
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`HorizontalBracketSpanner.`**`fuse`**(*spanner*)
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`HorizontalBracketSpanner.`**`index`**(*component*)
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`HorizontalBracketSpanner.pop()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

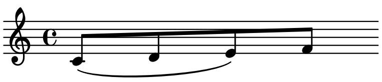
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

`HorizontalBracketSpanner.pop_left()`

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

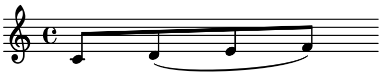
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

HorizontalBracketSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

HorizontalBracketSpanner.**__contains__**(*expr*)

HorizontalBracketSpanner.**__copy__**(**args*)

HorizontalBracketSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

HorizontalBracketSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HorizontalBracketSpanner.**__getitem__**(*expr*)

HorizontalBracketSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HorizontalBracketSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`HorizontalBracketSpanner.__len__()`

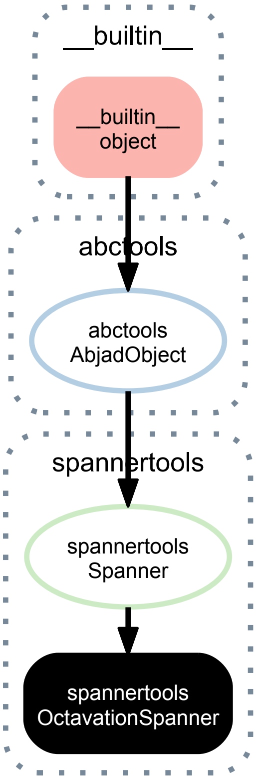
`HorizontalBracketSpanner.__lt__(expr)`
Trivial comparison to allow doctests to work.

`HorizontalBracketSpanner.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`HorizontalBracketSpanner.__repr__()`

33.2.10 spannertools.OctavationSpanner



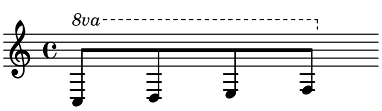
class `spannertools.OctavationSpanner`(*components=None, start=0, stop=0*)
Abjad octavation spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spanner = spannertools.OctavationSpanner(staff[:], start=1)
```

```
>>> f(staff)
\new Staff {
    \ottava #1
    c'8
    d'8
    e'8
    f'8
    \ottava #0
}
```

```
>>> show(staff)
```



Return octavation spanner.

Read-only properties

OctavationSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

OctavationSpanner.**duration**

Sum of prolated duration of all components in spanner.

OctavationSpanner.**duration_in_seconds**

Sum of duration of all leaves in spanner, in seconds.

OctavationSpanner.**leaves**

Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

OctavationSpanner.**override**

LilyPond grob override component plug-in.

OctavationSpanner.**preprolated_duration**

Sum of preprolated duration of all components in spanner.

OctavationSpanner.**set**

LilyPond context setting component plug-in.

OctavationSpanner.**storage_format**

Storage format of Abjad object.

Return string.

OctavationSpanner.**timespan**

Read-only timespan of spanner.

OctavationSpanner.**written_duration**

Sum of written duration of all components in spanner.

Read/write properties

OctavationSpanner.**start**

Get octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(staff[:], start=1)
>>> octavation.start
1
```

Set octavation start:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(staff[:], start=1)
>>> octavation.start
1
```

Set integer.

OctavationSpanner.**stop**

Get octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(staff[:], start=2, stop=1)
>>> octavation.stop
1
```

Set octavation stop:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> octavation = spannertools.OctavationSpanner(staff[:], start=2, stop=1)
>>> octavation.stop = 0
>>> octavation.stop
0
```

Set integer.

Methods

OctavationSpanner.**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

OctavationSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

OctavationSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

OctavationSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`OctavationSpanner.`**`extend_left`**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`OctavationSpanner.`**`fracture`**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

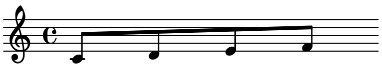
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

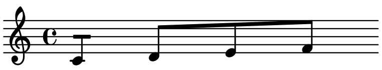
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`OctavationSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`OctavationSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`OctavationSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

OctavationSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

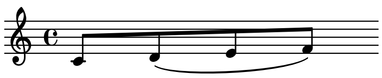
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

OctavationSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

OctavationSpanner.**__contains__**(*expr*)

OctavationSpanner.**__copy__**(**args*)

OctavationSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

OctavationSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OctavationSpanner.**__getitem__**(*expr*)

OctavationSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

OctavationSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OctavationSpanner.**__len__**()

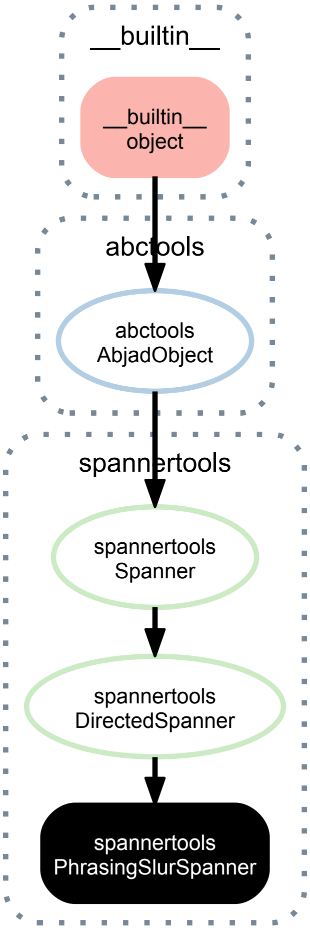
OctavationSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

OctavationSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

OctavationSpanner.**__repr__**()

33.2.11 spannertools.PhrasingSlurSpanner



class spannertools.**PhrasingSlurSpanner**(*components=None*, *direction=None*)

Abjad phrasing slur spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.PhrasingSlurSpanner(staff[:])
PhrasingSlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \(
    d'8
    e'8
    f'8 \)
}
```

```
>>> show(staff)
```

Return phrasing slur spanner.

Read-only properties

PhrasingSlurSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`PhrasingSlurSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`PhrasingSlurSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`PhrasingSlurSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`PhrasingSlurSpanner.`**`override`**
LilyPond grob override component plug-in.

`PhrasingSlurSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`PhrasingSlurSpanner.`**`set`**
LilyPond context setting component plug-in.

`PhrasingSlurSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`PhrasingSlurSpanner.`**`timespan`**
Read-only timespan of spanner.

`PhrasingSlurSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

`PhrasingSlurSpanner.`**`direction`**

Methods

`PhrasingSlurSpanner.`**`append`**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`PhrasingSlurSpanner.`**`append_left`**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

PhrasingSlurSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

PhrasingSlurSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

PhrasingSlurSpanner.**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

PhrasingSlurSpanner.**fracture**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

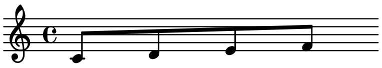
Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
```

```
        f'8 ]
}
```

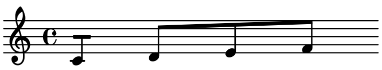
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`PhrasingSlurSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`PhrasingSlurSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

PhrasingSlurSpanner.**pop**()
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

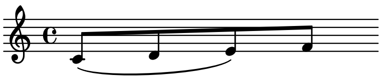
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

PhrasingSlurSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

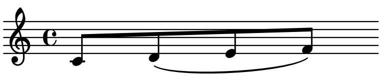
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

PhrasingSlurSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

PhrasingSlurSpanner.**__contains__**(*expr*)

PhrasingSlurSpanner.**__copy__**(**args*)

PhrasingSlurSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

PhrasingSlurSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PhrasingSlurSpanner.**__getitem__**(*expr*)

PhrasingSlurSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PhrasingSlurSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PhrasingSlurSpanner.**__len__**()

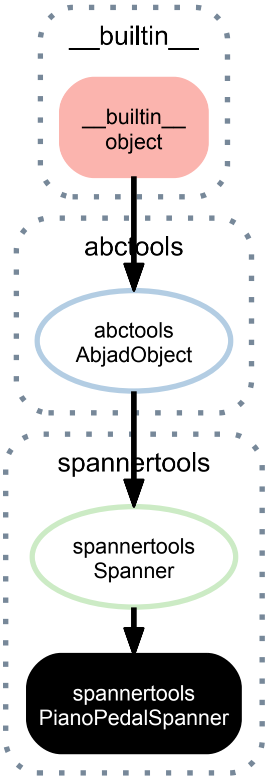
PhrasingSlurSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

PhrasingSlurSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

PhrasingSlurSpanner.**__repr__**()

33.2.12 spannertools.PianoPedalSpanner



class spannertools.**PianoPedalSpanner**(*components=None*)
Abjad piano pedal spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.PianoPedalSpanner(staff[:])
PianoPedalSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    \set Staff.pedalSustainStyle = #'mixed
    c'8 \sustainOn
    d'8
    e'8
    f'8 \sustainOff
}
```

```
>>> show(staff)
```

Return piano pedal spanner.

Read-only properties

`PianoPedalSpanner.`**`components`**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`PianoPedalSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`PianoPedalSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`PianoPedalSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`PianoPedalSpanner.`**`override`**
LilyPond grob override component plug-in.

`PianoPedalSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`PianoPedalSpanner.`**`set`**
LilyPond context setting component plug-in.

`PianoPedalSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`PianoPedalSpanner.`**`timespan`**
Read-only timespan of spanner.

`PianoPedalSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

`PianoPedalSpanner.`**`kind`**
Get piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.kind
'sustain'
```

Set piano pedal spanner kind:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.kind = 'sostenuto'
>>> spanner.kind
'sostenuto'
```

Acceptable values `'sustain'`, `'sostenuto'`, `'corda'`.

PianoPedalSpanner.**style**

Get piano pedal spanner style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.style
'mixed'
```

Set piano pedal spanner style:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.PianoPedalSpanner(staff[:])
>>> spanner.style = 'bracket'
>>> spanner.style
'bracket'
```

Acceptable values `'mixed'`, `'bracket'`, `'text'`.

Methods

PianoPedalSpanner.**append**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

PianoPedalSpanner.**append_left**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

PianoPedalSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

PianoPedalSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`PianoPedalSpanner.`**`extend_left`**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`PianoPedalSpanner.`**`fracture`**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

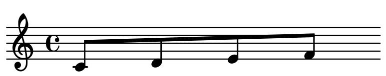
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

Set *direction=None* to fracture on both left and right sides.

Return tuple.

`PianoPedalSpanner.`**`fuse`**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

PianoPedalSpanner.**index**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

PianoPedalSpanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

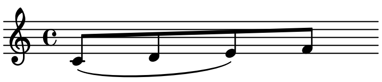
```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d'8
        e'8 )
        f'8
    }
```

```
>>> show(voice)
```



Return component.

PianoPedalSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

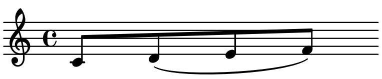
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

PianoPedalSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

PianoPedalSpanner.**__contains__**(*expr*)

PianoPedalSpanner.**__copy__**(**args*)

PianoPedalSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

PianoPedalSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PianoPedalSpanner.**__getitem__**(*expr*)

PianoPedalSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PianoPedalSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PianoPedalSpanner.**__len__**()

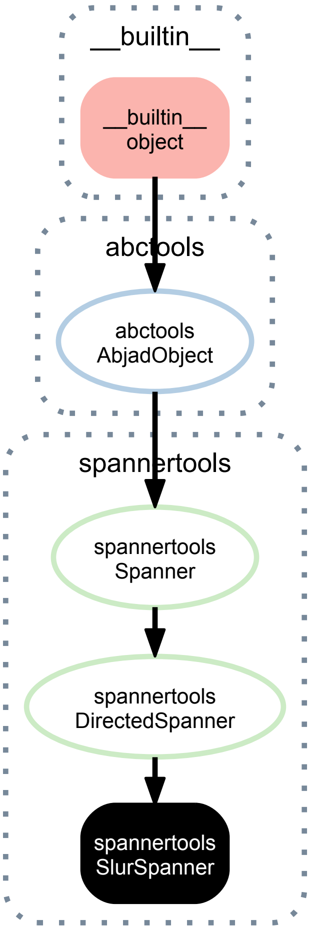
PianoPedalSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

PianoPedalSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

PianoPedalSpanner.**__repr__**()

33.2.13 spannertools.SlurSpanner



class spannertools.**SlurSpanner**(*components=None*, *direction=None*)

Abjad slur spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.SlurSpanner(staff[:])
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(staff)
```

Return slur spanner.

Read-only properties

SlurSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`SlurSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`SlurSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`SlurSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`SlurSpanner.`**`override`**
LilyPond grob override component plug-in.

`SlurSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`SlurSpanner.`**`set`**
LilyPond context setting component plug-in.

`SlurSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`SlurSpanner.`**`timespan`**
Read-only timespan of spanner.

`SlurSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

`SlurSpanner.`**`direction`**

Methods

`SlurSpanner.`**`append`**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`SlurSpanner.`**`append_left`**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`SlurSpanner.`**`clear`**`()`

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`SlurSpanner.`**`extend`**`(`*components*`)`

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`SlurSpanner.`**`extend_left`**`(`*components*`)`

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`SlurSpanner.`**`fracture`**`(`*i*, *direction=None*`)`

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

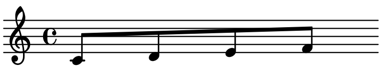
Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
```

```
        f'8 ]
    }
```

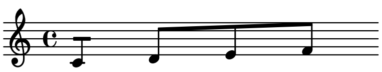
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`SlurSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`SlurSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

SlurSpanner.**pop**()

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

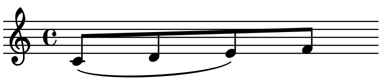
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

SlurSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

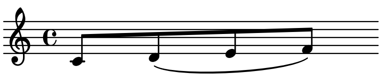
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

SlurSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

SlurSpanner.**__contains__**(*expr*)

SlurSpanner.**__copy__**(**args*)

SlurSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

SlurSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SlurSpanner.**__getitem__**(*expr*)

SlurSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SlurSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SlurSpanner.**__len__**()

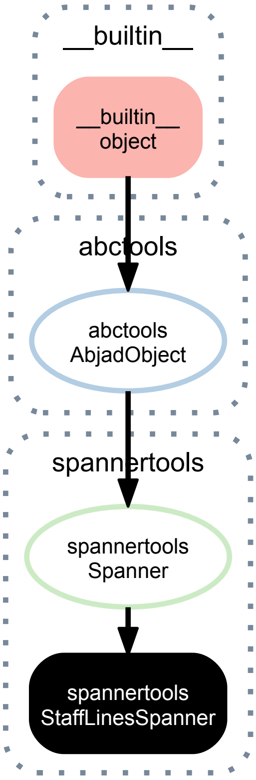
SlurSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

SlurSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SlurSpanner.**__repr__**()

33.2.14 spannertools.StaffLinesSpanner



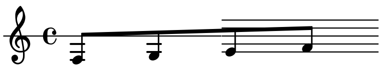
class spannertools.**StaffLinesSpanner**(*components=None*, *lines=5*)
Abjad staff lines spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.StaffLinesSpanner(staff[:2], 1)
StaffLinesSpanner(c'8, d'8)
```

```
>>> f(staff)
\new Staff {
    \stopStaff
    \override Staff.StaffSymbol #'line-count = #1
    \startStaff
    c'8
    d'8
    \stopStaff
    \revert Staff.StaffSymbol #'line-count
    \startStaff
    e'8
    f'8
}
```

```
>>> show(staff)
```



Staff lines spanner handles changing either the line-count or the line-positions property of the StaffSymbol grob, as well as automatically stopping and restarting the staff so that the change may take place.

Return staff lines spanner.

Read-only properties

StaffLinesSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

StaffLinesSpanner.**duration**
Sum of prolated duration of all components in spanner.

StaffLinesSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

StaffLinesSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

StaffLinesSpanner.**override**
LilyPond grob override component plug-in.

StaffLinesSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

StaffLinesSpanner.**set**
LilyPond context setting component plug-in.

StaffLinesSpanner.**storage_format**
Storage format of Abjad object.

Return string.

StaffLinesSpanner.**timespan**
Read-only timespan of spanner.

StaffLinesSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

StaffLinesSpanner.**lines**
Get staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(staff[:2], 1)
>>> spanner.lines
1
```

Set staff lines spanner line count:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.StaffLinesSpanner(staff[:2], 1)
>>> spanner.lines = 2
>>> spanner.lines
2
```

Set integer.

Methods

`StaffLinesSpanner.`**`append`**(*component*)

Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`StaffLinesSpanner.`**`append_left`**(*component*)

Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`StaffLinesSpanner.`**`clear`**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`StaffLinesSpanner.`**`extend`**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

StaffLinesSpanner.**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

StaffLinesSpanner.**fracture**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

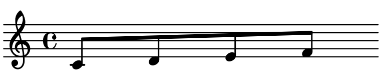
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

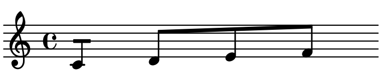
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

StaffLinesSpanner.**fuse**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`StaffLinesSpanner.`**`index`**`(`*component*`)`
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`StaffLinesSpanner.`**`pop`**`()`
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

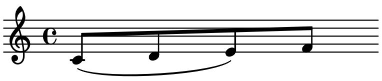
```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
```

```
        d'8
        e'8 )
        f'8
    }
```

```
>>> show(voice)
```



Return component.

StaffLinesSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

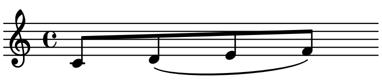
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

StaffLinesSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

`StaffLinesSpanner.`**`__contains__`**`(`*expr*`)`

`StaffLinesSpanner.`**`__copy__`**`(`**args*`)`

`StaffLinesSpanner.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`StaffLinesSpanner.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`StaffLinesSpanner.`**`__getitem__`**`(`*expr*`)`

`StaffLinesSpanner.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`StaffLinesSpanner.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`StaffLinesSpanner.`**`__len__`**`()`

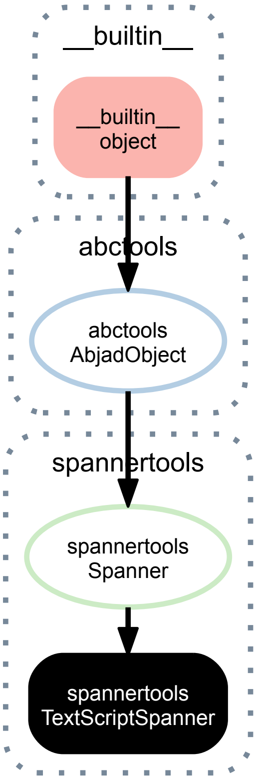
`StaffLinesSpanner.`**`__lt__`**`(`*expr*`)`
Trivial comparison to allow doctests to work.

`StaffLinesSpanner.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`StaffLinesSpanner.`**`__repr__`**`()`

33.2.15 spannertools.TextScriptSpanner



class spannertools.**TextScriptSpanner**(*components=None*)

New in version 2.0. Abjad text script spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spanner = spannertools.TextScriptSpanner(staff[:])
>>> spanner.override.text_script.color = 'red'
>>> markuptools.Markup(r'\italic { espressivo }', Up)(staff[1])
Markup((MarkupCommand('italic', ['espressivo']),), direction=Up)(d'8)
```

```
>>> f(staff)
\new Staff {
    \override TextScript #'color = #red
    c'8
    d'8 ^ \markup { \italic { espressivo } }
    e'8
    f'8
    \revert TextScript #'color
}
```

```
>>> show(staff)
```

Override LilyPond TextScript grob.

Return text script spanner.

Read-only properties

TextScriptSpanner.**components**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TextScriptSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`TextScriptSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`TextScriptSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TextScriptSpanner.`**`override`**
LilyPond grob override component plug-in.

`TextScriptSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`TextScriptSpanner.`**`set`**
LilyPond context setting component plug-in.

`TextScriptSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`TextScriptSpanner.`**`timespan`**
Read-only timespan of spanner.

`TextScriptSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Methods

`TextScriptSpanner.`**`append`**`(`*component*`)`
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`TextScriptSpanner.`**`append_left`**`(`*component*`)`
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

TextScriptSpanner.**clear**()

Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

TextScriptSpanner.**extend**(*components*)

Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

TextScriptSpanner.**extend_left**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

TextScriptSpanner.**fracture**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

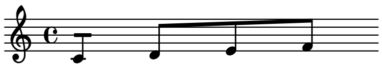
```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```

```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`TextScriptSpanner.`**`fuse`**(*spanner*)

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`TextScriptSpanner.`**`index`**(*component*)

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`TextScriptSpanner.pop()`
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

`TextScriptSpanner.pop_left()`
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

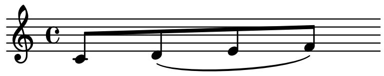
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

TextScriptSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

TextScriptSpanner.**__contains__**(*expr*)

TextScriptSpanner.**__copy__**(**args*)

TextScriptSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

TextScriptSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TextScriptSpanner.**__getitem__**(*expr*)

TextScriptSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TextScriptSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TextScriptSpanner.**__len__**()

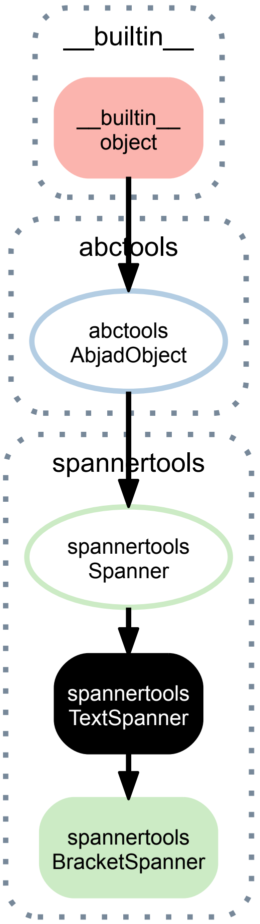
TextScriptSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

TextScriptSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TextScriptSpanner.**__repr__**()

33.2.16 spannertools.TextSpanner



class spannertools.**TextSpanner**(*components=None*)
New in version 2.0. Abjad text spanner:

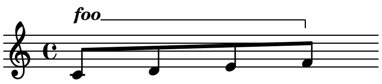
```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> text_spanner = spannertools.TextSpanner(staff[:])
```

```
>>> markup = markuptools.Markup(markuptools.MarkupCommand(
...     'bold', markuptools.MarkupCommand('italic', 'foo')))
>>> text_spanner.override.text_spanner.bound_details__left__text = markup
>>> markup = markuptools.Markup(
...     markuptools.MarkupCommand('draw-line', schemetools.SchemePair(0, -1)))
>>> text_spanner.override.text_spanner.bound_details__right__text = markup
>>> text_spanner.override.text_spanner.dash_fraction = 1
```

```
>>> f(staff)
\new Staff {
    \override TextSpanner #'bound-details #'left #'text = \markup {
        \bold \italic foo }
    \override TextSpanner #'bound-details #'right #'text = \markup {
        \draw-line #'(0 . -1) }
```

```
        \override TextSpanner #'dash-fraction = #1
        c'8 \startTextSpan
        d'8
        e'8
        f'8 \stopTextSpan
        \revert TextSpanner #'bound-details #'left #'text
        \revert TextSpanner #'bound-details #'right #'text
        \revert TextSpanner #'dash-fraction
    }
```

```
>>> show(staff)
```



Override LilyPond TextSpanner grob.

Return text spanner.

Read-only properties

`TextSpanner.`**`components`**

Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TextSpanner.`**`duration`**

Sum of prolated duration of all components in spanner.

`TextSpanner.`**`duration_in_seconds`**

Sum of duration of all leaves in spanner, in seconds.

`TextSpanner.`**`leaves`**

Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TextSpanner.`**`override`**

LilyPond grob override component plug-in.

`TextSpanner.`**`preprolated_duration`**

Sum of preprolated duration of all components in spanner.

`TextSpanner.`**`set`**

LilyPond context setting component plug-in.

`TextSpanner.`**`storage_format`**

Storage format of Abjad object.

Return string.

`TextSpanner.`**`timespan`**

Read-only timespan of spanner.

`TextSpanner.`**`written_duration`**

Sum of written duration of all components in spanner.

Methods

`TextSpanner.append(component)`
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

`TextSpanner.append_left(component)`
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

`TextSpanner.clear()`
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

`TextSpanner.extend(components)`
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`TextSpanner.extend_left(components)`
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

TextSpanner.**fracture**(*i*, *direction=None*)
Fracture spanner at *direction* of component at index *i*.

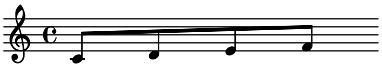
Valid values for *direction* are Left, Right and None.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

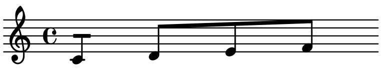
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

TextSpanner.**fuse**(*spanner*)
Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
```

```
        d'8
        e'8
        f'8 ]
    }
```

Return list.

`TextSpanner.index(component)`
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`TextSpanner.pop()`
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

`TextSpanner.pop_left()`
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

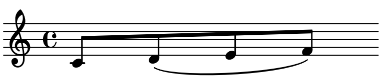
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

TextSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

TextSpanner.**__contains__**(*expr*)

TextSpanner.**__copy__**(**args*)

TextSpanner.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

TextSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TextSpanner.**__getitem__**(*expr*)

TextSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TextSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TextSpanner.**__len__**()

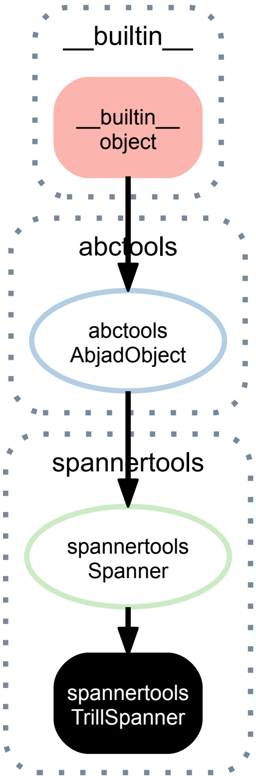
TextSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

TextSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TextSpanner.**__repr__**()

33.2.17 spannertools.TrillSpanner



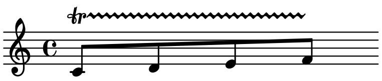
class spannertools.**TrillSpanner**(*components=None*)
Abjad trill spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.TrillSpanner(staff[:])
TrillSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 \startTrillSpan
    d'8
    e'8
    f'8 \stopTrillSpan
}
```

```
>>> show(staff)
```



Override LilyPond TrillSpanner grob.

Return trill spanner.

Read-only properties

TrillSpanner.**components**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

TrillSpanner.**duration**
Sum of prolated duration of all components in spanner.

TrillSpanner.**duration_in_seconds**
Sum of duration of all leaves in spanner, in seconds.

TrillSpanner.**leaves**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

TrillSpanner.**override**
LilyPond grob override component plug-in.

TrillSpanner.**preprolated_duration**
Sum of preprolated duration of all components in spanner.

TrillSpanner.**set**
LilyPond context setting component plug-in.

TrillSpanner.**storage_format**
Storage format of Abjad object.

Return string.

TrillSpanner.**timespan**
Read-only timespan of spanner.

TrillSpanner.**written_duration**
Sum of written duration of all components in spanner.

Read/write properties

TrillSpanner.**pitch**
Optional read / write pitch for pitched trills.

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> trill = spannertools.TrillSpanner(t[:2])
>>> trill.pitch = pitchtools.NamedChromaticPitch('cs', 4)
```

```
>>> f(t)
\new Staff {
    \pitchedTrill c'8 \startTrillSpan cs'
    d'8 \stopTrillSpan
    e'8
    f'8
}
```

Set pitch.

TrillSpanner.**written_pitch**

Methods

TrillSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

TrillSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

TrillSpanner.**clear**()
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

TrillSpanner.**extend**(*components*)
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`TrillSpanner.`**`extend_left`**(*components*)

Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`TrillSpanner.`**`fracture`**(*i*, *direction=None*)

Fracture spanner at *direction* of component at index *i*.

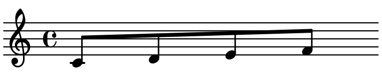
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

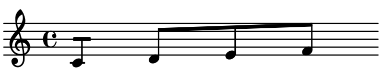
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`TrillSpanner.`**`fuse`**`(`*spanner*`)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
    e'8 [
    f'8 ]
}
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`TrillSpanner.`**`index`**`(`*component*`)`

Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`TrillSpanner.`**`pop`**`()`

Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

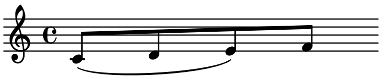
```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```



Return component.

TrillSpanner.**pop_left**()
Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

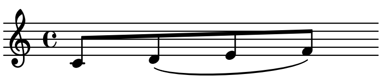
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

TrillSpanner.**__call__**(*expr*)
New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

TrillSpanner.**__contains__**(*expr*)

TrillSpanner.**__copy__**(**args*)

TrillSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

TrillSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TrillSpanner.**__getitem__**(*expr*)

TrillSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TrillSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TrillSpanner.**__len__**()

TrillSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

TrillSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TrillSpanner.**__repr__**()

33.3 Functions

33.3.1 spannertools.all_are_spanners

spannertools.**all_are_spanners**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad spanners:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
```

```
>>> spannertools.all_are_spanners([spanner])
True
```

True when *expr* is an empty sequence:

```
>>> spannertools.all_are_spanners([])
True
```

Otherwise false:

```
>>> spannertools.all_are_spanners('foo')
False
```

Return boolean.

33.3.2 spannertools.apply_octavation_spanner_to_pitched_components

`spannertools.`**`apply_octavation_spanner_to_pitched_components`**(*expr*, *ottava_numbered_diatonic_pitch=None*, *quindecisima_numbered_diatonic_pitch=None*)

New in version 1.1. Apply octavation spanner to pitched components in *expr*:

```
>>> t = Measure((4, 8), notetools.make_notes([24, 26, 27, 29], [(1, 8)]))
>>> spannertools.apply_octavation_spanner_to_pitched_components(
...     t, ottava_numbered_diatonic_pitch=14)
OctavationSpanner(|4/8(4)|)
```

```
>>> f(t)
    {
        \time 4/8
        \ottava #1
        c'''8
        d'''8
        ef'''8
        f'''8
        \ottava #0
    }
```

Apply octavation spanner according to the diatonic pitch number of the maximum pitch in *expr*.

Return octavation spanner.

33.3.3 spannertools.destroy_spanners_attached_to_component

`spannertools.`**`destroy_spanners_attached_to_component`**(*component*, *klass=None*)

New in version 1.1. Destroy spanners of *klass* attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> spanners = spannertools.destroy_spanners_attached_to_component(staff[0])
```

```
>>> f(staff)
\new Staff {
    c'8 \startTrillSpan
    d'8
    e'8
    f'8 \stopTrillSpan
}
```

Destroy all spanners when *klass* is none.

Return tuple of zero or more empty spanners.

Order of spanners in return value can not be predicted.

33.3.4 spannertools.destroy_spanners_attached_to_components_in_expr

spannertools.**destroy_spanners_attached_to_components_in_expr**(*expr*, *klass=None*)

New in version 2.9. Destroy spanners of *klass* attached to components in *expr*:

```
>>> staff = Staff("c'4 [ ( d' e' f' ) ]")
```

```
>>> f(staff)
\new Staff {
    c'4 [ (
    d'4
    e'4
    f'4 ] )
}
```

```
>>> spanners = spannertools.destroy_spanners_attached_to_components_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
}
```

Return tuple of zero or more empty spanners.

Order of spanners in return value can not be predicted.

33.3.5 spannertools.find_index_of_spanner_component_at_score_offset

spannertools.**find_index_of_spanner_component_at_score_offset**(*spanner*, *score_offset*)

Return index of component in 'spanner' that begins at exactly 'score_offset':

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> spannertools.find_index_of_spanner_component_at_score_offset(beam, Duration(3, 8))
3
```

Raise spanner population error when no component in *spanner* begins at exactly *score_offset*.

33.3.6 spannertools.find_spanner_component_starting_at_exactly_score_offset

spannertools.**find_spanner_component_starting_at_exactly_score_offset**(*spanner*, *score_offset*)

Find *spanner* component starting at exactly *score_offset*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> spannertools.find_spanner_component_starting_at_exactly_score_offset(
...     beam, Duration(3, 8))
Note("f'8")
```

When no *spanner* component starts at exactly *score_offset* return none.

Return *spanner* component or none.

33.3.7 spannertools.fracture_spanners_attached_to_component

spannertools.**fracture_spanners_attached_to_component**(*component*, *direction=None*, *klass=None*)

New in version 1.1. Fracture all spanners attached to *component* according to *direction*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> parts = spannertools.fracture_spanners_attached_to_component(staff[1], Right)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8 ] )
    e'8 [ (
    f'8 ] ) \stopTrillSpan
}
```

Set *direction* to `Left`, `Right` or `None`.

33.3.8 spannertools.fracture_spanners_that_cross_components

spannertools.**fracture_spanners_that_cross_components**(*components*)

Fracture to the left of the leftmost component. Fracture to the right of the rightmost component. Do not fracture spanners of any components at higher levels of score. Do not fracture spanners of any components at lower levels of score. Return components.

Components must be thread-contiguous. Some spanners may copy during fracture. This helper is public-safe.

Example:

```
>>> t = Staff(Container(notetools.make_repeated_notes(2)) * 3)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(t)
>>> crescendo = spannertools.CrescendoSpanner(t)
>>> beam = beamtools.BeamSpanner(t[:])
>>> trill = spannertools.TrillSpanner(t.leaves)

>>> f(t)
\new Staff {
    {
        c'8 [ \< \startTrillSpan
        d'8
    }
    {
        e'8
        f'8
    }
    {
        g'8
        a'8 ] \! \stopTrillSpan
    }   }

>>> spannertools.fracture_spanners_that_cross_components(t[1:2])
Selection({e'8, f'8},)

>>> f(t)
\new Staff {
    {
        c'8 [ \< \startTrillSpan
        d'8 ]
    }
    {
        e'8 [
        f'8 ]
    }
    {
        g'8 [
        a'8 ] \! \stopTrillSpan
    }
}
```

Return selection.

33.3.9 spannertools.get_nth_leaf_in_spanner

`spannertools.`**`get_nth_leaf_in_spanner`**(*spanner*, *idx*)

Get nth leaf in spanner, no matter how complicated the nesting situation.

Return leaf.

33.3.10 spannertools.get_spanners_attached_to_any_improper_child_of_component

`spannertools.`**`get_spanners_attached_to_any_improper_child_of_component`**(*component*, *klass=None*)

New in version 2.0. Get all spanners attached to any improper children of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> first_slur = spannertools.SlurSpanner(staff.leaves[:2])
>>> second_slur = spannertools.SlurSpanner(staff.leaves[2:])
>>> trill = spannertools.TrillSpanner(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8 )
    e'8 (
    f'8 ] ) \stopTrillSpan
}
```

```
>>> len(spannertools.get_spanners_attached_to_any_improper_child_of_component(staff))
4
```

Get all spanners of *klass* attached to any proper children of *component*:

```
>>> spanner_klass = spannertools.SlurSpanner
>>> result = spannertools.get_spanners_attached_to_any_proper_child_of_component(
... staff, spanner_klass)
```

```
>>> list(sorted(result))
[SlurSpanner(c'8, d'8), SlurSpanner(e'8, f'8)]
```

Get all spanners of any *klass* attached to any proper children of *component*:

```
>>> spanner_klasses = (spannertools.SlurSpanner, beamtools.BeamSpanner)
>>> result = spannertools.get_spanners_attached_to_any_proper_child_of_component(
... staff, spanner_klasses)
```

```
>>> list(sorted(result))
[BeamSpanner(c'8, d'8, e'8, f'8), SlurSpanner(c'8, d'8), SlurSpanner(e'8, f'8)]
```

Return unordered set of zero or more spanners.

33.3.11 spannertools.get_spanners_attached_to_any_improper_parent_of_component

spannertools.**get_spanners_attached_to_any_improper_parent_of_component**(*component*, *klass=None*)

New in version 1.1. Get all spanners attached to improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> result = list(sorted(
... spannertools.get_spanners_attached_to_any_improper_parent_of_component(staff[0])))
```

```
>>> for spanner in result:
...     spanner
...
BeamSpanner(c'8, d'8, e'8, f'8)
SlurSpanner(c'8, d'8, e'8, f'8)
TrillSpanner({c'8, d'8, e'8, f'8})
```

Return unordered set of zero or more spanners.

33.3.12 spannertools.get_spanners_attached_to_any_proper_child_of_component

spannertools.**get_spanners_attached_to_any_proper_child_of_component**(*component*, *klass=None*)

New in version 2.0. Get all spanners attached to any proper children of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> first_slur = spannertools.SlurSpanner(staff.leaves[:2])
>>> second_slur = spannertools.SlurSpanner(staff.leaves[2:])
>>> trill = spannertools.TrillSpanner(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8 )
    e'8 (
    f'8 ] ) \stopTrillSpan
}
```

```
>>> len(spannertools.get_spanners_attached_to_any_proper_child_of_component(
...     staff))
3
```

Get all spanners of *klass* attached to any proper children of *component*:

```
>>> spanner_klass = spannertools.SlurSpanner
>>> result = spannertools.get_spanners_attached_to_any_proper_child_of_component(
...     staff, spanner_klass)
```

```
>>> list(sorted(result))
[SlurSpanner(c'8, d'8), SlurSpanner(e'8, f'8)]
```

Get all spanners of any *klass* attached to any proper children of *component*:

```
>>> spanner_klasses = (spannertools.SlurSpanner, beamtools.BeamSpanner)
>>> result = spannertools.get_spanners_attached_to_any_proper_child_of_component(
...     staff, spanner_klasses)
```

```
>>> list(sorted(result))
[BeamSpanner(c'8, d'8, e'8, f'8), SlurSpanner(c'8, d'8), SlurSpanner(e'8, f'8)]
```

Return unordered set of zero or more spanners.

33.3.13 spannertools.get_spanners_attached_to_any_proper_parent_of_component

spannertools.**get_spanners_attached_to_any_proper_parent_of_component**(*component*, *klass=None*)

New in version 2.0. Get all spanners attached to any proper parent of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> spannertools.get_spanners_attached_to_any_proper_parent_of_component(staff[0])
set([TrillSpanner({c'8, d'8, e'8, f'8})])
```

Return unordered set of zero or more spanners.

33.3.14 spannertools.get_spanners_attached_to_component

spannertools.**get_spanners_attached_to_component**(*component*, *klass=None*)

New in version 2.0. Get all spanners attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> first_slur = spannertools.SlurSpanner(staff.leaves[:2])
>>> second_slur = spannertools.SlurSpanner(staff.leaves[2:])
>>> crescendo = spannertools.CrescendoSpanner(staff.leaves)
```

```
>>> f(staff)
\new Staff {
    c'8 [ \< (
    d'8 )
    e'8 (
    f'8 ] \! )
}
```

```
>>> result = spannertools.get_spanners_attached_to_component(staff.leaves[0])
>>> for x in sorted(result):
...     x
...
BeamSpanner(c'8, d'8, e'8, f'8)
CrescendoSpanner(c'8, d'8, e'8, f'8)
SlurSpanner(c'8, d'8)
```

Get spanners of *klass* attached to *component*:

```
>>> klass = beamtools.BeamSpanner
>>> result = spannertools.get_spanners_attached_to_component(staff.leaves[0], klass)
>>> for x in sorted(result):
...     x
...
BeamSpanner(c'8, d'8, e'8, f'8)
```

Get spanners of any *klass* attached to *component*:

```
>>> klasses = (beamtools.BeamSpanner, spannertools.SlurSpanner)
>>> result = spannertools.get_spanners_attached_to_component(staff.leaves[0], klasses)
>>> for x in sorted(result):
...     x
...
BeamSpanner(c'8, d'8, e'8, f'8)
SlurSpanner(c'8, d'8)
```

Return unordered set of zero or more spanners.

33.3.15 spannertools.get_spanners_contained_by_components

`spannertools.`**`get_spanners_contained_by_components`**(*components*)

Return unordered set of spanners contained within any component in list of thread-contiguous components. Getter for t.spanners.contained across thread-contiguous components.

33.3.16 spannertools.get_spanners_covered_by_components

`spannertools.`**`get_spanners_covered_by_components`**(*components*)

New in version 1.1. Get spanners covered by *components*.

Return unordered set of spanners completely contained within the time bounds of thread-contiguous components.

A spanner *p* is covered by timespan *t* when and only when `t.start_ofset <= p.timespan.start_offset and p.timespan.stop_offset <= t.timespan.stop_offset`.

33.3.17 spannertools.get_spanners_on_components_or_component_children

`spannertools.`**`get_spanners_on_components_or_component_children`**(*components*)

Return unordered set of all spanners attaching to any component in *components* or attaching to any of the children of any of the components in *components*.

33.3.18 spannertools.get_spanners_that_cross_components

spannertools.**get_spanners_that_cross_components**(*components*)

Assert thread-contiguous components. Collect spanners that attach to any component in 'components'. Return unordered set of crossing spanners. A spanner P crosses a list of thread-contiguous components C when P and C share at least one component and when it is the case that NOT ALL of the components in P are also in C. In other words, there is some intersection – but not total intersection – between the components of P and C.

Compare 'crossing' spanners with 'covered' spanners. Compare 'crossing' spanners with 'dominant' spanners. Compare 'crossing' spanners with 'contained' spanners. Compare 'crossing' spanners with 'attached' spanners.

Return spanners.

33.3.19 spannertools.get_spanners_that_dominate_component_pair

spannertools.**get_spanners_that_dominate_component_pair**(*left*, *right*)

Return Python list of (spanner, index) pairs. 'left' must be either an Abjad component or None. 'right' must be either an Abjad component or None.

If both 'left' and 'right' are components, then 'left' and 'right' must be thread-contiguous.

This is a special version of spannertools.get_spanners_that_dominate_components(). This version is useful for finding spanners that dominant a zero-length 'crack' between components, as in t[2:2].

Return spanners.

33.3.20 spannertools.get_spanners_that_dominate_components

spannertools.**get_spanners_that_dominate_components**(*components*)

Return Python list of (spanner, index) pairs. Each (spanner, index) pair gives a spanner which dominates all components in 'components' together with the start-index at which spanner first encounters 'components'.

Use this helper to 'lift' any and all spanners temporarily from 'components', perform some action to the underlying score tree, and then reattach all spanners to new score components.

This operation always leaves all expressions in tact.

33.3.21 spannertools.get_spanners_that_dominate_container_components_from_to

spannertools.**get_spanners_that_dominate_container_components_from_to**(*container*, *start*, *stop*)

Return Python list of (spanner, index) pairs. Each spanner dominates the components specified by slice with start index 'start' and stop index 'stop'. Generalization of dominant spanner-finding functions for slices. This exists for slices like t[2:2] that are empty lists.

Return spanners.

33.3.22 spannertools.get_the_only_spanner_attached_to_any_improper_parent_of_compon

spannertools.**get_the_only_spanner_attached_to_any_improper_parent_of_component**(*component*, *klass=None*)

New in version 1.1. Get the only spanner attached to any improper parent *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
```

```
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> print spannertools.get_the_only_spanner_attached_to_component(staff)
TrillSpanner({c'8, d'8, e'8, f'8})
```

Raise missing spanner error when no spanner attached to *component*.

Raise extra spanner error when more than one spanner attached to *component*.

Return a single spanner.

Note: function will usually be called with *klass* specifier set.

33.3.23 spannertools.get_the_only_spanner_attached_to_component

spannertools.**get_the_only_spanner_attached_to_component**(*component*, *klass=None*)

New in version 1.1. Get the only spanner attached to *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> slur = spannertools.SlurSpanner(staff.leaves)
>>> trill = spannertools.TrillSpanner(staff)
>>> f(staff)
\new Staff {
    c'8 [ ( \startTrillSpan
    d'8
    e'8
    f'8 ] ) \stopTrillSpan
}
```

```
>>> print spannertools.get_the_only_spanner_attached_to_component(staff)
TrillSpanner({c'8, d'8, e'8, f'8})
```

Raise missing spanner error when no spanner attached to *component*.

Raise extra spanner error when more than one spanner attached to *component*.

Return a single spanner.

Note: function will usually be called with *klass* specifier set.

33.3.24 spannertools.is_component_with_spanner_attached

spannertools.**is_component_with_spanner_attached**(*expr*, *klass=None*)

New in version 2.0. True when *expr* is a component with spanner attached:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(staff.leaves)
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

```
>>> spannertools.is_component_with_spanner_attached(staff[0])
True
```

Otherwise false:

```
>>> spannertools.is_component_with_spanner_attached(staff)
False
```

When *klass* is not none then true when *expr* is a component with a spanner of *klass* attached.

Return true or false.

33.3.25 spannertools.iterate_components_in_spanner

spannertools.**iterate_components_in_spanner**(*spanner*, *klass=None*, *reverse=False*)
New in version 2.10. Yield components in *spanner* one at a time from left to right:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> p = beamtools.BeamSpanner(t[2:])
```

```
>>> notes = spannertools.iterate_components_in_spanner(p, klass=Note)
```

```
>>> for note in notes:
...   note
Note("e'8")
Note("f'8")
```

Yield components in *spanner* one at a time from right to left:

```
>>> notes = spannertools.iterate_components_in_spanner(p, klass=Note, reverse=True)
```

```
>>> for note in notes:
...   note
Note("f'8")
Note("e'8")
```

Return generator.

33.3.26 spannertools.make_covered_spanner_schema

spannertools.**make_covered_spanner_schema**(*components*)
New in version 2.0. Make schema of spanners covered by *components*:

```
>>> voice = Voice(Measure((2, 8), notetools.make_repeated_notes(2)) * 4)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
>>> slur = spannertools.SlurSpanner(voice[-2:])
```

```
>>> f(voice)
\new Voice {
    {
        \time 2/8
        c'8 [
        d'8
    }
    {
        e'8
        f'8 ]
    }
    {
        g'8 (
        a'8
    }
    {
        b'8
        c''8 )
```

```
    }
}
```

```
>>> spannertools.make_covered_spanner_schema([voice])
{BeamSpanner(c'8, d'8, e'8, f'8): [2, 3, 5, 6], SlurSpanner(|2/8(2)|, |2/8(2)|): [7, 10]}
```

Return dictionary.

33.3.27 spannertools.make_dynamic_spanner_below_with_nib_at_right

spannertools.**make_dynamic_spanner_below_with_nib_at_right**(*dynamic_text*, *components=None*)

New in version 2.0. Span *components* with text spanner. Position spanner below staff and configure with *dynamic_text*, solid line and upward-pointing nib at right:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.make_dynamic_spanner_below_with_nib_at_right('mp', staff[:])
TextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    \override TextSpanner #'bound-details #'left #'text = \markup { \dynamic { mp } }
    \override TextSpanner #'bound-details #'right #'text = \markup { \draw-line #'(0 . 1) }
    \override TextSpanner #'bound-details #'right-broken #'text = ##f
    \override TextSpanner #'dash-fraction = #1
    \override TextSpanner #'direction = #down
    c'8 \startTextSpan
    d'8
    e'8
    f'8 \stopTextSpan
    \revert TextSpanner #'bound-details #'left #'text
    \revert TextSpanner #'bound-details #'right #'text
    \revert TextSpanner #'bound-details #'right-broken #'text
    \revert TextSpanner #'dash-fraction
    \revert TextSpanner #'direction
}
```

```
>>> show(staff)
```

Return spanner.

33.3.28 spannertools.make_solid_text_spanner_above_with_nib_at_right

spannertools.**make_solid_text_spanner_above_with_nib_at_right**(*left_text*, *components=None*)

New in version 2.0. Span *components* with text spanner. Position spanner above staff and configure with *left_text*, solid line and downward-pointing nib at right:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.make_solid_text_spanner_above_with_nib_at_right('foo', staff[:])
TextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    \override TextSpanner #'bound-details #'left #'text = \markup { foo }
    \override TextSpanner #'bound-details #'right #'text = \markup {
        \draw-line #'(0 . -1) }
    \override TextSpanner #'bound-details #'right-broken #'text = ##f
```

```
        \override TextSpanner #'dash-fraction = #1
        \override TextSpanner #'direction = #up
        c'8 \startTextSpan
        d'8
        e'8
        f'8 \stopTextSpan
        \revert TextSpanner #'bound-details #'left #'text
        \revert TextSpanner #'bound-details #'right #'text
        \revert TextSpanner #'bound-details #'right-broken #'text
        \revert TextSpanner #'dash-fraction
        \revert TextSpanner #'direction
    }
```

```
>>> show(staff)
```

Return spanner.

33.3.29 spannertools.make_solid_text_spanner_below_with_nib_at_right

`spannertools.`**`make_solid_text_spanner_below_with_nib_at_right`**(*left_text*, *components=None*)

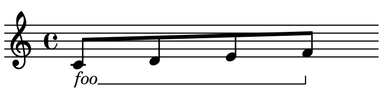
New in version 2.0. Span *components* with text spanner. Position spanner below staff and configure with *left_text*, solid line and upward-pointing nib at right:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> spannertools.make_solid_text_spanner_below_with_nib_at_right(
...     'foo', staff[:])
TextSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    \override TextSpanner #'bound-details #'left #'text = \markup { foo }
    \override TextSpanner #'bound-details #'right #'text = \markup { \draw-line #'(0 . 1) }
    \override TextSpanner #'bound-details #'right-broken #'text = ##f
    \override TextSpanner #'dash-fraction = #1
    \override TextSpanner #'direction = #down
    c'8 \startTextSpan
    d'8
    e'8
    f'8 \stopTextSpan
    \revert TextSpanner #'bound-details #'left #'text
    \revert TextSpanner #'bound-details #'right #'text
    \revert TextSpanner #'bound-details #'right-broken #'text
    \revert TextSpanner #'dash-fraction
    \revert TextSpanner #'direction
}
```

```
>>> show(staff)
```



Return spanner.

33.3.30 spannertools.make_spanner_schema

`spannertools.`**`make_spanner_schema`**(*components*)

New in version 2.0. Make schema of spanners contained by *components*:

```
>>> voice = Voice(Measure((2, 8), notetools.make_repeated_notes(2)) * 4)
>>> pitchtools.set_ascending_named_diatonic_pitches_on_tie_chains_in_expr(voice)
>>> beam = beamtools.BeamSpanner(voice.leaves[:4])
>>> slur = spannertools.SlurSpanner(voice[-2:])
```

```
>>> f(voice)
\new Voice {
    {
        \time 2/8
        c'8 [
        d'8
    }
    {
        e'8
        f'8 ]
    }
    {
        g'8 (
        a'8
    }
    {
        b'8
        c''8 )
    }
}
```

```
>>> spannertools.make_spanner_schema(voice.leaves[2:4])
{BeamSpanner(c'8, d'8, e'8, f'8): [0, 1]}
```

Return dictionary.

33.3.31 spannertools.move_spanners_from_component_to_children_of_component

spannertools.**move_spanners_from_component_to_children_of_component**(*donor*)

Give spanners attaching directly to donor to recipients. Usual use is to give attached spanners from parent to children, which is a composer-safe operation.

Return none.

33.3.32 spannertools.withdraw_components_from_spanners_covered_by_components

spannertools.**withdraw_components_from_spanners_covered_by_components**(*components*)

Find every spanner covered by ‘components’. Withdraw all components in ‘components’ from covered spanners. Return ‘components’. The operation always leaves all score trees in tact.

Return components.

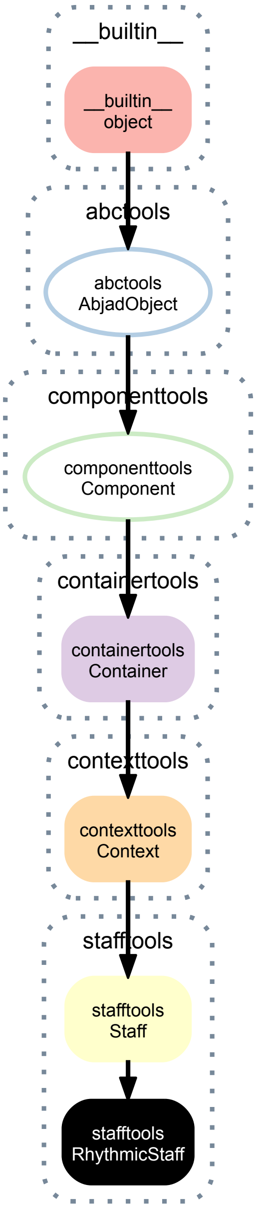
CHAPTER

THIRTYFOUR

STAFFTOOLS

34.1 Concrete Classes

34.1.1 stafftools.RhythmicStaff

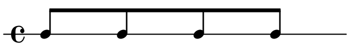


class stafftools.**RhythmicStaff**(*music=None*, *context_name='RhythmicStaff'*, *name=None*)
Abjad model of a rhythmic staff:

```
>>> staff = stafftools.RhythmicStaff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new RhythmicStaff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```



Return RhythmicStaff instance.

Read-only properties

RhythmicStaff.**contents_duration**

RhythmicStaff.**descendants**
Read-only reference to component descendants score selection.

RhythmicStaff.**duration**

RhythmicStaff.**duration_in_seconds**

RhythmicStaff.**engraver_consists**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

RhythmicStaff.**engraver_removals**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

RhythmicStaff.**is_semantic**

RhythmicStaff.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

RhythmicStaff.**lilypond_format**

`RhythmicStaff.`**lineage**
Read-only reference to component lineage score selection.

`RhythmicStaff.`**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`RhythmicStaff.`**override**
Read-only reference to LilyPond grob override component plug-in.

`RhythmicStaff.`**parent**

`RhythmicStaff.`**parentage**
Read-only reference to component parentage score selection.

`RhythmicStaff.`**preprolated_duration**

`RhythmicStaff.`**prolation**

`RhythmicStaff.`**set**
Read-only reference LilyPond context setting component plug-in.

`RhythmicStaff.`**spanners**
Read-only reference to unordered set of spanners attached to component.

`RhythmicStaff.`**storage_format**
Storage format of Abjad object.

Return string.

`RhythmicStaff.`**timespan**
Read-only timespan of component.

`RhythmicStaff.`**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

`RhythmicStaff.`**context_name**
Read / write name of context as a string.

`RhythmicStaff.`**is_nonsemantic**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
```

```
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

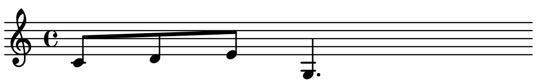
RhythmicStaff.**is_parallel**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```

Return none.

`RhythmicStaff.name`
Read-write name of context. Must be string or none.

Methods

`RhythmicStaff.append(component)`
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

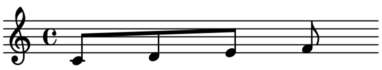
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`RhythmicStaff.extend(expr)`
Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

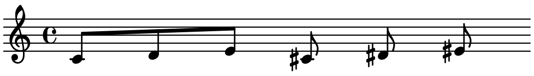
```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
```

```
        e'8 ]
        cs'8
        ds'8
        es'8
    }
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`RhythmicStaff.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`RhythmicStaff.`**`insert`**`(`*i*, *component*`)`

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

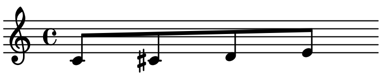
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`RhythmicStaff.`**`pop`**`(`*i=-1*`)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

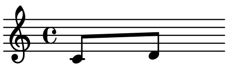
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`RhythmicStaff.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

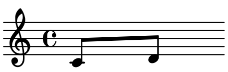
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

RhythmicStaff.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

RhythmicStaff.**__contains__**(*expr*)
True if expr is in container, otherwise False.

RhythmicStaff.**__copy__**(**args*)

RhythmicStaff.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

RhythmicStaff.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

RhythmicStaff.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmicStaff.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

RhythmicStaff.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

RhythmicStaff.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

RhythmicStaff.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

RhythmicStaff.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmicStaff.**__len__**()
Return nonnegative integer number of components in container.

RhythmicStaff.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RhythmicStaff.**__mul__**(*n*)

RhythmicStaff.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

RhythmicStaff.**__radd__**(*expr*)
Extend container by contents of expr to the right.

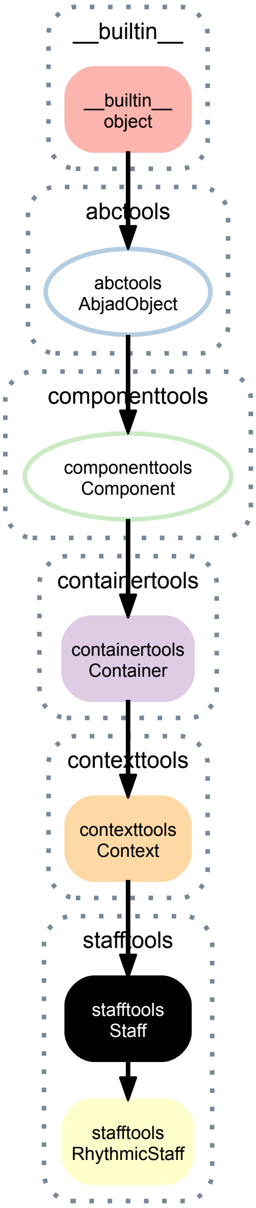
RhythmicStaff.**__repr__**()

RhythmicStaff.**__rmul__**(*n*)

RhythmicStaff.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate

self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

34.1.2 stafftools.Staff



class `stafftools.Staff` (*music=None*, *context_name='Staff'*, *name=None*)

Abjad model of a staff:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(staff)
```

Return Staff instance.

Read-only properties

Staff.**contents_duration**

Staff.**descendants**
Read-only reference to component descendants score selection.

Staff.**duration**

Staff.**duration_in_seconds**

Staff.**engraver_consists**
New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

Staff.**engraver_removals**
New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

Staff.**is_semantic**

Staff.**leaves**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

Staff.**lilypond_format**

Staff.**lineage**
Read-only reference to component lineage score selection.

Staff.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

`Staff.`**`override`**
Read-only reference to LilyPond grob override component plug-in.

`Staff.`**`parent`**

`Staff.`**`parentage`**
Read-only reference to component parentage score selection.

`Staff.`**`preprolated_duration`**

`Staff.`**`prolation`**

`Staff.`**`set`**
Read-only reference LilyPond context setting component plug-in.

`Staff.`**`spanners`**
Read-only reference to unordered set of spanners attached to component.

`Staff.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Staff.`**`timespan`**
Read-only timespan of component.

`Staff.`**`timespan_in_seconds`**
Read-only timespan of component in seconds.

Read/write properties

`Staff.`**`context_name`**
Read / write name of context as a string.

`Staff.`**`is_nonsemantic`**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
        s1 * 1/8
    }
    {
        \time 5/16
        s1 * 5/16
    }
    {
        s1 * 5/16
    }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

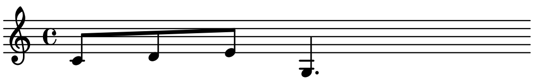
The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

Staff.**is_parallel**
Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```

Return none.

Staff.**name**
Read-write name of context. Must be string or none.

Methods

Staff.**append**(*component*)
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

`Staff.extend(expr)`
Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```

Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Staff.index(component)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Staff.insert(i, component)`

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

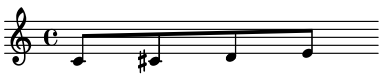
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Staff.pop(i=-1)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

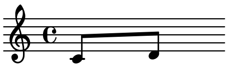
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Staff.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

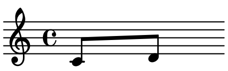
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Staff.`**`__add__`**`(`*expr*`)`

Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

`Staff.`**`__contains__`**`(`*expr*`)`

True if expr is in container, otherwise False.

`Staff.`**`__copy__`**`(`**args*`)`

`Staff.`**`__delitem__`**`(`*i*`)`
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

`Staff.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Staff.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Staff.`**`__getitem__`**`(`*i*`)`
Return component at index i in container. Shallow traversal of container for numeric indices only.

`Staff.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Staff.`**`__iadd__`**`(`*expr*`)`
__iadd__ avoids unnecessary copying of structures.

`Staff.`**`__imul__`**`(`*total*`)`
Multiply contents of container 'total' times. Return multiplied container.

`Staff.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Staff.`**`__len__`**`()`
Return nonnegative integer number of components in container.

`Staff.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Staff.`**`__mul__`**`(`*n*`)`

`Staff.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`Staff.`**`__radd__`**`(`*expr*`)`
Extend container by contents of expr to the right.

`Staff.`**`__repr__`**`()`

`Staff.`**`__rmul__`**`(`*n*`)`

`Staff.`**`__setitem__`**`(`*i*, *expr*`)`
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

34.2 Functions

34.2.1 stafftools.all_are_staves

`stafftools.`**`all_are_staves`**`(`*expr*`)`
New in version 2.6. True when *expr* is a sequence of Abjad staves:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
```

```
>>> stafftools.all_are_staves([staff])
True
```

True when *expr* is an empty sequence:

```
>>> stafftools.all_are_staves([])
True
```

Otherwise false:

```
>>> stafftools.all_are_staves('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

34.2.2 stafftools.get_first_staff_in_improper_parentage_of_component

stafftools.**get_first_staff_in_improper_parentage_of_component**(*component*)

New in version 2.0. Get first staff in improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> stafftools.get_first_staff_in_improper_parentage_of_component(staff[1])
Staff{4}
```

Return staff or none.

34.2.3 stafftools.get_first_staff_in_proper_parentage_of_component

stafftools.**get_first_staff_in_proper_parentage_of_component**(*component*)

New in version 2.0. Get first staff in proper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> stafftools.get_first_staff_in_proper_parentage_of_component(staff[1])
Staff{4}
```

Return staff or none.

34.2.4 stafftools.make_rhythmic_sketch_staff

stafftools.**make_rhythmic_sketch_staff**(*music*)

Make rhythmic staff with transparent time_signature and transparent bar lines.

CHAPTER

THIRTYFIVE

STRINGTOOLS

35.1 Functions

35.1.1 stringtools.arg_to_bidirectional_direction_string

stringtools.**arg_to_bidirectional_direction_string**(*arg*)

Convert *arg* to bidirectional direction string:

```
>>> stringtools.arg_to_bidirectional_direction_string('^')
'up'
```

```
>>> stringtools.arg_to_bidirectional_direction_string('_')
'down'
```

```
>>> stringtools.arg_to_bidirectional_direction_string(1)
'up'
```

```
>>> stringtools.arg_to_bidirectional_direction_string(-1)
'down'
```

If *arg* is 'up' or 'down', *arg* will be returned.

Return string or none.

35.1.2 stringtools.arg_to_bidirectional_lilypond_symbol

stringtools.**arg_to_bidirectional_lilypond_symbol**(*arg*)

Convert *arg* to bidirectional LilyPond symbol:

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Up)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Down)
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(1)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(-1)
'_'
```

If *arg* is '^' or '_', *arg* will be returned.

Return str or None.

35.1.3 stringtools.arg_to_tridirectional_direction_string

stringtools.**arg_to_tridirectional_direction_string**(*arg*)

Convert *arg* to tridirectional direction string:

```
>>> stringtools.arg_to_tridirectional_direction_string('^')
'up'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('-')
'center'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('_')
'down'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(1)
'up'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(0)
'center'
```

```
>>> stringtools.arg_to_tridirectional_direction_string(-1)
'down'
```

```
>>> stringtools.arg_to_tridirectional_direction_string('default')
'center'
```

If *arg* is None, None will be returned.

Return str or None.

35.1.4 stringtools.arg_to_tridirectional_lilypond_symbol

stringtools.**arg_to_tridirectional_lilypond_symbol**(*arg*)

Convert *arg* to tridirectional LilyPond symbol:

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Up)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol('neutral')
'-'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol('default')
'-'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(Down)
'_'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(1)
'^'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(0)
'-'
```

```
>>> stringtools.arg_to_tridirectional_lilypond_symbol(-1)
'_'
```

If *arg* is None, None will be returned.

If *arg* is '^', '-', or '_', *arg* will be returned.

Return string or None.

35.1.5 stringtools.arg_to_tridirectional_ordinal_constant

stringtools.**arg_to_tridirectional_ordinal_constant**(*arg*)

Convert *arg* to tridirectional ordinal constant:

```
>>> stringtools.arg_to_tridirectional_ordinal_constant('^')
Up
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant('_')
Down
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant(1)
Up
```

```
>>> stringtools.arg_to_tridirectional_ordinal_constant(-1)
Down
```

If *arg* is Up, Center or Down, *arg* will be returned.

Return OrdinalConstant or None.

35.1.6 stringtools.capitalize_string_start

`stringtools.`**`capitalize_string_start`**`(`*string*`)`

New in version 2.5. Capitalize *string*:

```
>>> string = 'violin I'
```

```
>>> stringtools.capitalize_string_start(string)
'Violin I'
```

Function differs from built-in `string.capitalize()`.

This function affects only `string[0]` and leaves noninitial characters as-is.

Built-in `string.capitalize()` forces noninitial characters to lowercase.

```
>>> string.capitalize()
'Violin i'
```

Return newly constructed string.

35.1.7 stringtools.format_input_lines_as_doc_string

`stringtools.`**`format_input_lines_as_doc_string`**`(`*input_lines*`)`

New in version 2.0. Format *input_lines* as doc string.

Format expressions intelligently.

Treat blank lines intelligently.

Capture hash-suffixed line output.

Use when writing docstrings.

Example skipped because docstring goes crazy on example input.

35.1.8 stringtools.format_input_lines_as_regression_test

`stringtools.`**`format_input_lines_as_regression_test`**`(`*input_lines*, *tab_width=3*`)`

New in version 2.0. Format *input_lines* as regression test:

```
>>> input_lines = '''
... staff = Staff("c'8 d'8 e'8 f'8")
... beamtools.BeamSpanner(staff.leaves)
... f(staff)
...
... tuplettools.FixedDurationTuplet(Duration(2, 8), staff[:3])
... f(staff)
... '''
```

```
>>> stringtools.format_input_lines_as_regression_test(input_lines)

    staff = Staff("c'8 d'8 e'8 f'8")
    beamtools.BeamSpanner(staff.leaves)

    r'''
    \new Staff {
        c'8 [
        d'8
        e'8
        f'8 ]
    }
    '''

    tuplettools.FixedDurationTuplet(Duration(2, 8), staff[:3])

    r'''
    \new Staff {
        \times 2/3 {
            c'8 [
            d'8
            e'8
        }
        f'8 ]
    }

    assert wellformednesstools.is_well_formed_component(staff)
    assert staff.lilypond_format == "\\new Staff {
        \n\t\\times 2/3 {\n\t\tc'8 [\n\t\td'8\n\t\te'8\n\t}\n\tf'8 ]\n}"
    '''
```

Format expressions intelligently.

Treat blank lines intelligently.

Remove line-final hash characters.

Used when writing tests.

35.1.9 stringtools.is_lowercamelcase_string

stringtools.**is_lowercamelcase_string**(*expr*)

New in version 2.5. True when *expr* is a string and is lowercamelcase:

```
>>> stringtools.is_lowercamelcase_string('fooBar')
True
```

False otherwise:

```
>>> stringtools.is_lowercamelcase_string('FooBar')
False
```

Return boolean.

35.1.10 stringtools.is_space_delimited_lowercase_string

stringtools.**is_space_delimited_lowercase_string**(*expr*)

New in version 2.5. True when *expr* is a string and is space-delimited lowercase:

```
>>> stringtools.is_space_delimited_lowercase_string('foo bar')
True
```

False otherwise:

```
>>> stringtools.is_space_delimited_lowercase_string('foo_bar')
False
```

Return boolean.

35.1.11 stringtools.is_underscore_delimited_lowercase_file_name

stringtools.**is_underscore_delimited_lowercase_file_name**(*expr*)

New in version 2.7. True when *expr* is a string and is underscore-delimited lowercase file name with extension:

```
>>> stringtools.is_underscore_delimited_lowercase_file_name('foo_bar')
True
```

False otherwise:

```
>>> stringtools.is_underscore_delimited_lowercase_file_name('foo.bar.blah')
False
```

Return boolean.

35.1.12 stringtools.is_underscore_delimited_lowercase_file_name_with_extension

stringtools.**is_underscore_delimited_lowercase_file_name_with_extension**(*expr*)

New in version 2.7. True when *expr* is a string and is underscore-delimited lowercase file name with extension:

```
>>> stringtools.is_underscore_delimited_lowercase_file_name_with_extension('foo_bar.blah')
True
```

False otherwise:

```
>>> stringtools.is_underscore_delimited_lowercase_file_name_with_extension('foo.bar.blah')
False
```

Return boolean.

35.1.13 stringtools.is_underscore_delimited_lowercase_package_name

stringtools.**is_underscore_delimited_lowercase_package_name**(*expr*)

New in version 2.5. True when *expr* is a string and is underscore-delimited lowercase package name:

```
>>> stringtools.is_underscore_delimited_lowercase_package_name('foo.bar.blah_package')
True
```

False otherwise:

```
>>> stringtools.is_underscore_delimited_lowercase_package_name('foo.bar.BlahPackage')
False
```

Return boolean.

35.1.14 stringtools.is_underscore_delimited_lowercase_string

stringtools.**is_underscore_delimited_lowercase_string**(*expr*)

New in version 2.5. True when *expr* is a string and is underscore delimited lowercase:

```
>>> stringtools.is_underscore_delimited_lowercase_string('foo_bar')
True
```

False otherwise:

```
>>> stringtools.is_underscore_delimited_lowercase_string('foo bar')
False
```

Return boolean.

35.1.15 stringtools.is_uppercamelcase_string

stringtools.**is_uppercamelcase_string**(*expr*)

New in version 2.5. True when *expr* is a string and is uppercamelcase:

```
>>> stringtools.is_uppercamelcase_string('FooBar')
True
```

False otherwise:

```
>>> stringtools.is_uppercamelcase_string('fooBar')
False
```

Return boolean.

35.1.16 stringtools.space_delimited_lowercase_to_uppercamelcase

stringtools.**space_delimited_lowercase_to_uppercamelcase**(*string*)

New in version 2.6. Change space-delimited lowercase *string* to uppercamelcase:

```
>>> string = 'bass figure alignment positioning'
>>> stringtools.space_delimited_lowercase_to_uppercamelcase(string)
'BassFigureAlignmentPositioning'
```

Return string.

35.1.17 stringtools.string_to_strict_directory_name

stringtools.**string_to_strict_directory_name**(*string*)

New in version 2.6. Change *string* to strict directory name:

```
>>> stringtools.string_to_strict_directory_name('Déja vu')
'deja_vu'
```

Strip accents from accented characters. Change all punctuation (including spaces) to underscore. Set to lowercase.

Return string.

35.1.18 stringtools.strip_diacritics_from_binary_string

stringtools.**strip_diacritics_from_binary_string**(*binary_string*)

New in version 2.5. Strip diacritics from *binary_string*:

```
>>> binary_string = 'Dvo\xc5\x99\xc3\xa1k'
```

```
>>> print binary_string
Dvořák
```

```
>>> stringtools.strip_diacritics_from_binary_string(binary_string)
'Dvorak'
```

Return ASCII string.

35.1.19 stringtools.underscore_delimited_lowercase_to_lowercamelcase

stringtools.**underscore_delimited_lowercase_to_lowercamelcase**(*string*)

New in version 2.0. Change underscore-delimited lowercase *string* to lowercamelcase:

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.underscore_delimited_lowercase_to_lowercamelcase(string)
'bassFigureAlignmentPositioning'
```

Return string.

35.1.20 stringtools.underscore_delimited_lowercase_to_uppercamelcase

stringtools.**underscore_delimited_lowercase_to_uppercamelcase**(*string*)
New in version 2.0. Change underscore-delimited lowercase *string* to uppercamelcase:

```
>>> string = 'bass_figure_alignment_positioning'
>>> stringtools.underscore_delimited_lowercase_to_uppercamelcase(string)
'BassFigureAlignmentPositioning'
```

Return string.

35.1.21 stringtools.uppercamelcase_to_space_delimited_lowercase

stringtools.**uppercamelcase_to_space_delimited_lowercase**(*string*)
New in version 2.6. Change uppercamelcase *string* to space-delimited lowercase:

```
>>> string = 'KeySignatureMark'
```

```
>>> stringtools.uppercamelcase_to_space_delimited_lowercase(string)
'key signature mark'
```

Return string.

35.1.22 stringtools.uppercamelcase_to_underscore_delimited_lowercase

stringtools.**uppercamelcase_to_underscore_delimited_lowercase**(*string*)
New in version 2.6. Change uppercamelcase *string* to underscore-delimited lowercase:

```
>>> string = 'KeySignatureMark'
```

```
>>> stringtools.uppercamelcase_to_underscore_delimited_lowercase(string)
'key_signature_mark'
```

Return string.

CHAPTER

THIRTYSIX

TEMPOTOOLS

36.1 Functions

36.1.1 tempotools.report_integer_tempo_rewrite_pairs

tempotools.**report_integer_tempo_rewrite_pairs**(*integer_tempo*, *maximum_numerator=None*, *maximum_denominator=None*)

New in version 2.0. Report *integer_tempo* rewrite pairs.

Allow no tempo less than half *integer_tempo* or greater than double *integer_tempo*:

```
>>> tempotools.report_integer_tempo_rewrite_pairs(
...    58, maximum_numerator=8, maximum_denominator=8)
2:1      29
1:1      58
2:3      87
1:2      116
```

With more lenient numerator and denominator:

```
>>> tempotools.report_integer_tempo_rewrite_pairs(
...      58, maximum_numerator=30, maximum_denominator=30)
2:1      29
29:15    30
29:16    32
29:17    34
29:18    36
29:19    38
29:20    40
29:21    42
29:22    44
29:23    46
29:24    48
29:25    50
29:26    52
29:27    54
29:28    56
1:1      58
29:30    60
2:3      87
1:2      116
```

Return none.

36.1.2 tempotools.rewrite_duration_under_new_tempo

tempotools.**rewrite_duration_under_new_tempo**(*duration*, *tempo_mark_1*, *tempo_mark_2*)

New in version 2.0. Rewrite prolated *duration* under new tempo.

Given prolated *duration* governed by *tempo_mark_1*, return new duration governed by *tempo_mark_2*.

Ensure that *duration* and new duration consume exactly the same amount of time in seconds.

Example. Consider the two tempo indications below.

```
>>> tempo_mark_1 = contexttools.TempoMark(Duration(1, 4), 60)
>>> tempo_mark_2 = contexttools.TempoMark(Duration(1, 4), 90)
```

The first tempo indication specifies quarter equal to `60 MM`.

The second tempo indication specifies quarter equal to `90 MM`.

The second tempo is `3/2` times as fast as the first:

```
>>> tempo_mark_2 / tempo_mark_1
Multiplier(3, 2)
```

Note that a triplet eighth note *tempo_mark_1* equals a regular eighth note under *tempo_mark_2*:

```
>>> tempotools.rewrite_duration_under_new_tempo(
...     Duration(1, 12), tempo_mark_1, tempo_mark_2)
Duration(1, 8)
```

And note that a regular eighth note under *tempo_mark_1* equals a dotted sixteenth under *tempo_mark_2*:

```
>>> tempotools.rewrite_duration_under_new_tempo(
...     Duration(1, 8), tempo_mark_1, tempo_mark_2)
Duration(3, 16)
```

Return duration.

36.1.3 tempotools.rewrite_integer_tempo

`tempotools.`**`rewrite_integer_tempo`**`(`*integer_tempo*, *maximum_numerator=None*, *maximum_denominator=None*`)`

New in version 2.0. Rewrite *integer_tempo*.

Allow no tempo less than half *integer_tempo* or greater than double *integer_tempo*:

```
>>> pairs = tempotools.rewrite_integer_tempo(
...     58, maximum_numerator=8, maximum_denominator=8)
```

```
>>> for pair in pairs:
...     pair
...
(Multiplier(1, 2), 29)
(Multiplier(1, 1), 58)
(Multiplier(3, 2), 87)
(Multiplier(2, 1), 116)
```

Return list.

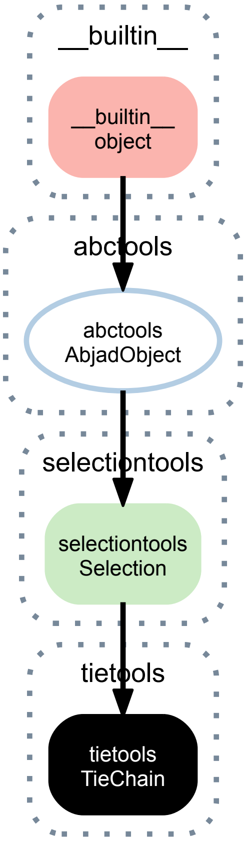
CHAPTER

THIRTYSEVEN

TIETOOLS

37.1 Concrete Classes

37.1.1 tietools.TieChain



class `tietools.TieChain`(*music*)

New in version 2.9. All the notes in a tie chain:

```
>>> staff = Staff("c' d' e' ~ e'")
```

```
>>> tietools.get_tie_chain(staff[2])
TieChain(Note("e'4"), Note("e'4"))
```

Tie chains are immutable score selections.

Read-only properties

`TieChain.`**`all_leaves_are_in_same_parent`**
True when all leaves in tie chain are in same parent.

Return boolean.

`TieChain.`**`duration_in_seconds`**
Read-only duration in seconds of components in tie chain.

Return duration.

`TieChain.`**`head`**
Read-only reference to element `0` in tie chain.

`TieChain.`**`is_pitched`**
True when tie chain head is a note or chord.

Return boolean.

`TieChain.`**`is_trivial`**
True when length of tie chain is less than or equal to `1`.

Return boolean.

`TieChain.`**`leaves`**
Read-only tuple of leaves in tie spanner.

`TieChain.`**`leaves_grouped_by_immediate_parents`**
Read-only list of leaves in tie chain grouped by immediate parents of leaves.

Return list of lists.

`TieChain.`**`music`**
Read-only tuple of components in selection.

`TieChain.`**`preprolated_duration`**
Sum of preprolated durations of all components in tie chain.

`TieChain.`**`prolated_duration`**
Sum of prolated durations of all components in tie chain.

`TieChain.`**`storage_format`**
Storage format of Abjad object.

Return string.

`TieChain.`**`timespan`**
Read-only timespan of selection.

`TieChain.`**`written_duration`**
Sum of written duration of all components in tie chain.

Special methods

`TieChain.`**`__add__`**(*expr*)

`TieChain.`**`__contains__`**(*expr*)

`TieChain.`**`__eq__`**(*expr*)

`TieChain.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`TieChain.`**`__getitem__`**(*expr*)

`TieChain.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TieChain.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TieChain.__len__()`

`TieChain.__lt__(expr)`
Abjad objects by default do not implement this method.

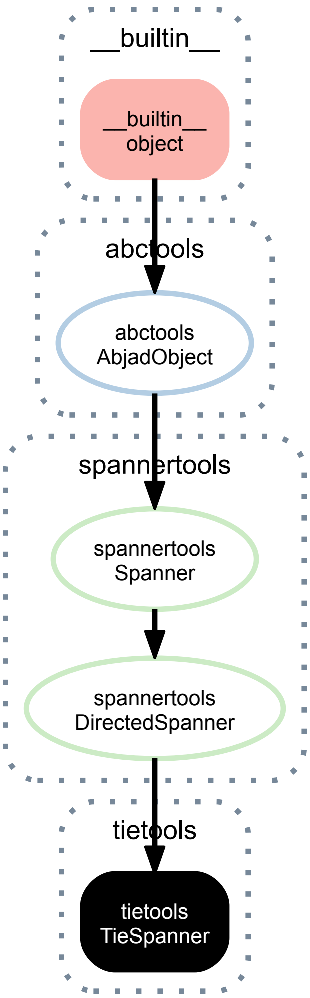
Raise exception.

`TieChain.__ne__(expr)`

`TieChain.__radd__(expr)`

`TieChain.__repr__()`

37.1.2 tietools.TieSpanner

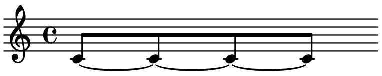


class `tietools.TieSpanner(music=None, direction=None)`
Abjad tie spanner:

```
>>> staff = Staff(notetools.make_repeated_notes(4))
>>> tietools.TieSpanner(staff[:])
TieSpanner(c'8, c'8, c'8, c'8)
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8 ~
    c'8 ~
    c'8
}
```

```
>>> show(staff)
```



Return tie spanner.

Read-only properties

`TieSpanner.`**`components`**
Return read-only tuple of components in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.components
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TieSpanner.`**`duration`**
Sum of prolated duration of all components in spanner.

`TieSpanner.`**`duration_in_seconds`**
Sum of duration of all leaves in spanner, in seconds.

`TieSpanner.`**`leaves`**
Return read-only tuple of leaves in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner.leaves
(Note("c'8"), Note("d'8"))
```

Return tuple.

`TieSpanner.`**`override`**
LilyPond grob override component plug-in.

`TieSpanner.`**`preprolated_duration`**
Sum of preprolated duration of all components in spanner.

`TieSpanner.`**`set`**
LilyPond context setting component plug-in.

`TieSpanner.`**`storage_format`**
Storage format of Abjad object.

Return string.

`TieSpanner.`**`timespan`**
Read-only timespan of spanner.

`TieSpanner.`**`written_duration`**
Sum of written duration of all components in spanner.

Read/write properties

TieSpanner.**direction**

Methods

TieSpanner.**append**(*component*)
Add *component* to right of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.append(voice[2])
>>> spanner
BeamSpanner(c'8, d'8, e'8)
```

Return none.

TieSpanner.**append_left**(*component*)
Add *component* to left of spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.append_left(voice[1])
>>> spanner
BeamSpanner(d'8, e'8, f'8)
```

Return none.

TieSpanner.**clear**()
Remove all components from spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> spanner.clear()
>>> spanner
BeamSpanner()
```

Return none.

TieSpanner.**extend**(*components*)
Add iterable *components* to right of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8)
```

```
>>> spanner.extend(voice[2:])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

TieSpanner.**extend_left**(*components*)
Add iterable *components* to left of spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.extend_left(voice[:2])
>>> spanner
BeamSpanner(c'8, d'8, e'8, f'8)
```

Return none.

`TieSpanner.fracture(i, direction=None)`

Fracture spanner at *direction* of component at index *i*.

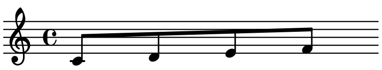
Valid values for *direction* are `Left`, `Right` and `None`.

Return original, left and right spanners.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> beam = beamtools.BeamSpanner(voice[:])
>>> beam
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

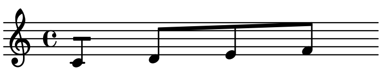
```
>>> show(voice)
```



```
>>> beam.fracture(1, direction=Left)
(BeamSpanner(c'8, d'8, e'8, f'8), BeamSpanner(c'8), BeamSpanner(d'8, e'8, f'8))
```

```
>>> f(voice)
\new Voice {
    c'8 [ ]
    d'8 [
    e'8
    f'8 ]
}
```

```
>>> show(voice)
```



Set *direction=None* to fracture on both left and right sides.

Return tuple.

`TieSpanner.fuse(spanner)`

Fuse contiguous spanners.

Return new spanner.

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> left_beam = beamtools.BeamSpanner(voice[:2])
>>> right_beam = beamtools.BeamSpanner(voice[2:])
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8 ]
```

```
        e'8 [
        f'8 ]
    }
```

```
>>> left_beam.fuse(right_beam)
[(BeamSpanner(c'8, d'8), BeamSpanner(e'8, f'8), BeamSpanner(c'8, d'8, e'8, f'8))]
```

```
>>> print voice.lilypond_format
\new Voice {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

Return list.

`TieSpanner.index(component)`
Return nonnegative integer index of *component* in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = beamtools.BeamSpanner(voice[2:])
>>> spanner
BeamSpanner(e'8, f'8)
```

```
>>> spanner.index(voice[-2])
0
```

Return nonnegative integer.

`TieSpanner.pop()`
Remove and return rightmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

```
>>> show(voice)
```

```
>>> spanner.pop()
Note("f'8")
```

```
>>> spanner
SlurSpanner(c'8, d'8, e'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8 )
    f'8
}
```

```
>>> show(voice)
```

Return component.

TieSpanner.**pop_left**()

Remove and return leftmost component in spanner:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> spanner = spannertools.SlurSpanner(voice[:])
>>> spanner
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8 (
    d'8
    e'8
    f'8 )
}
```

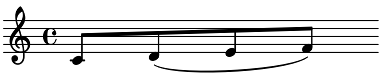
```
>>> show(voice)
```

```
>>> spanner.pop_left()
Note("c'8")
```

```
>>> spanner
SlurSpanner(d'8, e'8, f'8)
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8 (
    e'8
    f'8 )
}
```

```
>>> show(voice)
```



Return component.

Special methods

TieSpanner.**__call__**(*expr*)

New in version 2.9. Call spanner on *expr* as a shortcut to extend spanner:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
```

```
>>> beam = beamtools.BeamSpanner()
>>> beam(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

The method is provided as an experimental way of unifying spanner and mark attachment syntax.

Return spanner.

TieSpanner.**__contains__**(*expr*)

TieSpanner.**__copy__**(**args*)

TieSpanner.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

TieSpanner.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TieSpanner.**__getitem__**(*expr*)

TieSpanner.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TieSpanner.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TieSpanner.**__len__**()

TieSpanner.**__lt__**(*expr*)
Trivial comparison to allow doctests to work.

TieSpanner.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TieSpanner.**__repr__**()

37.2 Functions

37.2.1 tietools.add_or_remove_tie_chain_notes_to_achieve_scaled_written_duration

tietools.**add_or_remove_tie_chain_notes_to_achieve_scaled_written_duration**(*tie_chain*, *multiplier*)

Add or remove *tie_chain* notes to achieve scaled written duration:

```
>>> staff = Staff("c'8 [ ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [ ]
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff[0])
>>> tietools.add_or_remove_tie_chain_notes_to_achieve_scaled_written_duration(
...     tie_chain, Fraction(5, 4))
TieChain(Note("c'8"), Note("c'32"))
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'32 ]
}
```

Return *tie_chain*.

37.2.2 tietools.add_or_remove_tie_chain_notes_to_achieve_written_duration

tietools.**add_or_remove_tie_chain_notes_to_achieve_written_duration**(*tie_chain*, *new_written_duration*)

Add or remove *tie_chain* notes to achieve *written_duration*:

```
>>> staff = Staff("c'8 [ ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [ ]
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff[0])
>>> tietools.add_or_remove_tie_chain_notes_to_achieve_written_duration(
...     tie_chain, Duration(5, 32))
TieChain(Note("c'8"), Note("c'32"))
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'32 ]
}
```

Return *tie_chain*.

37.2.3 tietools.apply_tie_spanner_to_leaf_pair

tietools.**apply_tie_spanner_to_leaf_pair**(*left*, *right*)

Apply tie spanner to *left* leaf and *right* leaf:

```
>>> staff = Staff("c'8 ~ c' c' c'")
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8
    c'8
    c'8
}
```

```
>>> tietools.apply_tie_spanner_to_leaf_pair(staff[1], staff[2])
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8 ~
    c'8
    c'8
}
```

Handle existing tie spanners intelligently.

Return none.

37.2.4 tietools.are_components_in_same_tie_spanner

tietools.**are_components_in_same_tie_spanner**(*components*)

True when *components* are in same tie spanner:

```
>>> voice = Voice("c'8 ~ c' d' ~ d'")
```

```
>>> f(voice)
\new Voice {
    c'8 ~
    c'8
    d'8 ~
    d'8
}
```

```
>>> tietools.are_components_in_same_tie_spanner(voice[:2])
True
```

Otherwise false:

```
>>> tietools.are_components_in_same_tie_spanner(voice[1:3])
False
```

Return boolean.

37.2.5 tietools.get_nontrivial_tie_chains_masked_by_components

tietools.**get_nontrivial_tie_chains_masked_by_components**(*components*)

Get nontrivial tie chains masked by *components*:

```
>>> staff = Staff("c'8 ~ c'4 d'8 ~ d'4 e'4.")
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'4
    d'8 ~
    d'4
    e'4.
}
```

Return only nontrivial tie chains:

```
>>> tietools.get_nontrivial_tie_chains_masked_by_components(staff.leaves)
[TieChain(Note("c'8"), Note("c'4")), TieChain(Note("d'8"), Note("d'4"))]
```

Return 'masked' tie chains when only some notes of a tie chain are passed in:

```
>>> tietools.get_nontrivial_tie_chains_masked_by_components(staff.leaves[1:2])
[TieChain(Note("c'4"),)]
```

Return list of zero or more (possibly masked) tie chains.

37.2.6 tietools.get_tie_chain

tietools.**get_tie_chain**(*component*)

New in version 2.0. Get tie chain from *component*:

```
>>> staff = Staff("c'8 ~ c' d'4")
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8
    d'4
}
```

```
>>> tietools.get_tie_chain(staff[0])
TieChain(Note("c'8"), Note("c'8"))
```

Return tie chain.

37.2.7 tietools.get_tie_chains_masked_by_components

tietools.**get_tie_chains_masked_by_components**(*components*)
Get tie chains masked by *components*:

```
>>> staff = Staff("abj: | 2/4 c'4 d'4 ~ || 2/4 d'4 e'4 ~ || 2/4 e'4 f'4 |")
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'4
        d'4 ~
    }
    {
        d'4
        e'4 ~
    }
    {
        e'4
        f'4
    }
}
```

```
>>> for tie_chain in tietools.get_tie_chains_masked_by_components(staff.leaves):
...     tie_chain
...
TieChain(Note("c'4"),)
TieChain(Note("d'4"), Note("d'4"))
TieChain(Note("e'4"), Note("e'4"))
TieChain(Note("f'4"),)
```

```
>>> for tie_chain in tietools.get_tie_chains_masked_by_components(staff[1].leaves):
...     tie_chain
...
TieChain(Note("d'4"),)
TieChain(Note("e'4"),)
```

```
>>> for tie_chain in tietools.get_tie_chains_masked_by_components(staff[1:]):
...     tie_chain
...
TieChain(Note("d'4"),)
TieChain(Note("e'4"), Note("e'4"))
TieChain(Note("f'4"),)
```

Return list of zero or more (possibly masked) tie chains.

37.2.8 tietools.get_tie_spanner_attached_to_component

tietools.**get_tie_spanner_attached_to_component**(*component*)
New in version 2.10. Get the only tie spanner attached to *component*:

```
>>> staff = Staff("c'8 ~ c'8 d'4")
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8
    d'4
}
```

```
>>> tietools.get_tie_spanner_attached_to_component(staff[0])
TieSpanner(c'8, c'8)
```

Return tie spanner.

Raise missing spanner error when no tie spanner attached to *component*.

Raise extra spanner error when more than one tie spanner attached to *component*.

37.2.9 tietools.is_component_with_tie_spanner_attached

tietools.**is_component_with_tie_spanner_attached**(*expr*)

New in version 2.0. True when *expr* is component with tie spanner attached:

```
>>> staff = Staff("c'8 ~ c' ~ c' ~ c'")
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'8 ~
    c'8 ~
    c'8
}
```

```
>>> tietools.is_component_with_tie_spanner_attached(staff[1])
True
```

Otherwise false:

```
>>> tietools.is_component_with_tie_spanner_attached(staff)
False
```

Return boolean.

37.2.10 tietools.iterate_nontrivial_tie_chains_in_expr

tietools.**iterate_nontrivial_tie_chains_in_expr**(*expr*, *reverse=False*)

Iterate nontrivial tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 f'4 ~ f'16")
```

```
>>> f(staff)
\new Staff {
    c'4 ~
    \times 2/3 {
        c'16
        d'8
    }
    e'8
    f'4 ~
    f'16
}
```

```
>>> for x in tietools.iterate_nontrivial_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("f'4"), Note("f'16"))
```

Iterate nontrivial tie chains backward in *expr*:

```
>>> for x in tietools.iterate_nontrivial_tie_chains_in_expr(staff, reverse=True):
...     x
...
TieChain(Note("f'4"), Note("f'16"))
TieChain(Note("c'4"), Note("c'16"))
```

Return generator.

37.2.11 tietools.iterate_pitched_tie_chains_in_expr

`tietools.iterate_pitched_tie_chains_in_expr(expr, reverse=False)`
New in version 2.10. Iterate pitched tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 r8 f'8 ~ f'16 r8.")
```

```
>>> f(staff)
\new Staff {
    c'4 ~
    \times 2/3 {
        c'16
        d'8
    }
    e'8
    r8
    f'8 ~
    f'16
    r8.
}
```

```
>>> for x in tietools.iterate_pitched_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("d'8"),)
TieChain(Note("e'8"),)
TieChain(Note("f'8"), Note("f'16"))
```

Iterate pitched tie chains backward in *expr*:

```
::
```

```
>>> for x in tietools.iterate_pitched_tie_chains_in_expr(staff, reverse=True):
...     x
...
TieChain(Note("f'8"), Note("f'16"))
TieChain(Note("e'8"),)
TieChain(Note("d'8"),)
TieChain(Note("c'4"), Note("c'16"))
```

Tie chains are pitched if they comprise notes or chords.

Tie chains are not pitched if they comprise rests or skips.

Return generator.

37.2.12 tietools.iterate_tie_chains_in_expr

`tietools.iterate_tie_chains_in_expr(expr, reverse=False)`
Iterate tie chains forward in *expr*:

```
>>> staff = Staff(r"c'4 ~ \times 2/3 { c'16 d'8 } e'8 f'4 ~ f'16")
```

```
>>> f(staff)
\new Staff {
    c'4 ~
    \times 2/3 {
        c'16
        d'8
    }
    e'8
    f'4 ~
    f'16
}
```

```
>>> for x in tietools.iterate_tie_chains_in_expr(staff):
...     x
...
TieChain(Note("c'4"), Note("c'16"))
TieChain(Note("d'8"),)
TieChain(Note("e'8"),)
TieChain(Note("f'4"), Note("f'16"))
```

Iterate tie chains backward in *expr*:

```
::
```

```
>>> for x in tietools.iterate_tie_chains_in_expr(staff, reverse=True):
...     x
...
TieChain(Note("f'4"), Note("f'16"))
TieChain(Note("e'8"),)
TieChain(Note("d'8"),)
TieChain(Note("c'4"), Note("c'16"))
```

Return generator.

37.2.13 tietools.iterate_topmost_masked_tie_chains_and_containers_in_expr

tietools.**iterate_topmost_masked_tie_chains_and_containers_in_expr**(*expr*)

Iterate topmost tie chains and containers in *expr*, masked by *expr*:

```
>>> input = "abj: | 2/4 c'4 d'4 ~ |"
>>> input += "| 4/4 d'4 ~ 2/3 { d'4 d'4 d'4 } d'4 ~ |"
>>> input += "| 2/4 d'4 e'4 |"
>>> staff = Staff(input)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'4
        d'4 ~
    }
    {
        \time 4/4
        d'4 ~
        \times 2/3 {
            d'4
            d'4
            d'4
        }
        d'4 ~
    }
    {
        \time 2/4
        d'4
        e'4
    }
}
```

```
>>> for x in tietools.iterate_topmost_masked_tie_chains_and_containers_in_expr(
...     staff[0]): x
...
TieChain(Note("c'4"),)
TieChain(Note("d'4"),)
```

```
>>> for x in tietools.iterate_topmost_masked_tie_chains_and_containers_in_expr(
...     staff[1]): x
...
TieChain(Note("d'4"),)
Tuplet(2/3, [d'4, d'4, d'4])
TieChain(Note("d'4"),)
```

```
>>> for x in tietools.iterate_topmost_masked_tie_chains_and_containers_in_expr(
...     staff[2]): x
...
TieChain(Note("d'4"),)
TieChain(Note("e'4"),)
```

Return generator.

37.2.14 tietools.iterate_topmost_tie_chains_and_components_in_expr

tietools.**iterate_topmost_tie_chains_and_components_in_expr**(*expr*)

Iterate topemost tie chains and components forward in *expr*:

```
>>> string = r"c'8 ~ c'32 d'8 ~ d'32 \times 2/3 { e'8 f'8 g'8 } a'8 ~ a'32 b'8 ~ b'32"
>>> staff = Staff(string)
```

```
>>> f(staff)
\new Staff {
    c'8 ~
    c'32
    d'8 ~
    d'32
    \times 2/3 {
        e'8
        f'8
        g'8
    }
    a'8 ~
    a'32
    b'8 ~
    b'32
}
```

```
>>> for x in tietools.iterate_topmost_tie_chains_and_components_in_expr(staff):
...     x
...
TieChain(Note("c'8"), Note("c'32"))
TieChain(Note("d'8"), Note("d'32"))
Tuplet(2/3, [e'8, f'8, g'8])
TieChain(Note("a'8"), Note("a'32"))
TieChain(Note("b'8"), Note("b'32"))
```

Raise tie chain error on overlapping tie chains.

Return generator.

37.2.15 tietools.remove_nonfirst_leaves_in_tie_chain

tietools.**remove_nonfirst_leaves_in_tie_chain**(*tie_chain*)

Remove nonfirst leaves in *tie_chain*:

```
>>> staff = Staff("c'4 ~ c'16")
```

```
>>> f(staff)
\new Staff {
    c'4 ~
    c'16
}
```

```
>>> tietools.remove_nonfirst_leaves_in_tie_chain(tietools.get_tie_chain(staff[0]))
TieChain(Note("c'4"),)
```

```
>>> f(staff)
\new Staff {
    c'4
}
```

Return *tie_chain*.

37.2.16 tietools.remove_tie_spanners_from_components_in_expr

tietools.**remove_tie_spanners_from_components_in_expr**(*expr*)

Remove tie spanners components in *expr*:

```
>>> staff = Staff("c'4 ~ c'16 d'4 ~ d'16")
```

```
>>> f(staff)
\new Staff {
    c'4 ~
    c'16
    d'4 ~
    d'16
}
```

```
>>> tietools.remove_tie_spanners_from_components_in_expr(staff[:])
Selection(Note("c'4"), Note("c'16"), Note("d'4"), Note("d'16"))
```

```
>>> f(staff)
\new Staff {
    c'4
    c'16
    d'4
    d'16
}
```

Return *expr*.

37.2.17 tietools.tie_chain_to_tuplet_with_ratio

tietools.**tie_chain_to_tuplet_with_ratio**(*tie_chain*, *proportions*, *is_diminution=True*, *dotted=True*)

New in version 2.0. Example 1a. Change *tie_chain* to augmented tuplet with proportions `[1]`. Avoid dots:

```
>>> staff = Staff("c'8 [ ~ c'16 c'16 ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'16
    c'16 ]
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff[0])
>>> tietools.tie_chain_to_tuplet_with_ratio(
...     tie_chain, [1], is_diminution=False, dotted=False)
FixedDurationTuplet(3/16, [c'8])
```

```
>>> f(staff)
\new Staff {
    \fraction \times 3/2 {
        c'8 [
    }
    c'16 ]
}
```

Exampl 1b. Change *tie_chain* to augment tuplet with proportions `[1, 2]`. Avoid dots:

```
>>> staff = Staff("c'8 [ ~ c'16 c'16 ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'16
```

```
        c'16 ]
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff[0])
>>> tietools.tie_chain_to_tuplet_with_ratio(
...     tie_chain, [1, 2], is_diminution=False, dotted=False)
FixedDurationTuplet(3/16, [c'16, c'8])
```

```
>>> f(staff)
\new Staff {
    {
        c'16 [
        c'8
    }
    c'16 ]
}
```

Examle 1c. Change *tie_chain* to augmented tuplet with proportions `[1, 2, 2]`. Avoid dots:

```
>>> staff = Staff("c'8 [ ~ c'16 c'16 ]")
```

```
>>> f(staff)
\new Staff {
    c'8 [ ~
    c'16
    c'16 ]
}
```

```
>>> tie_chain = tietools.get_tie_chain(staff[0])
>>> tietools.tie_chain_to_tuplet_with_ratio(
...     tie_chain, [1, 2, 2], is_diminution=False, dotted=False)
FixedDurationTuplet(3/16, [c'32, c'16, c'16])
```

```
>>> f(staff)
\new Staff {
    \fraction \times 6/5 {
        c'32 [
        c'16
        c'16
    }
    c'16 ]
}
```

Return tuplet.

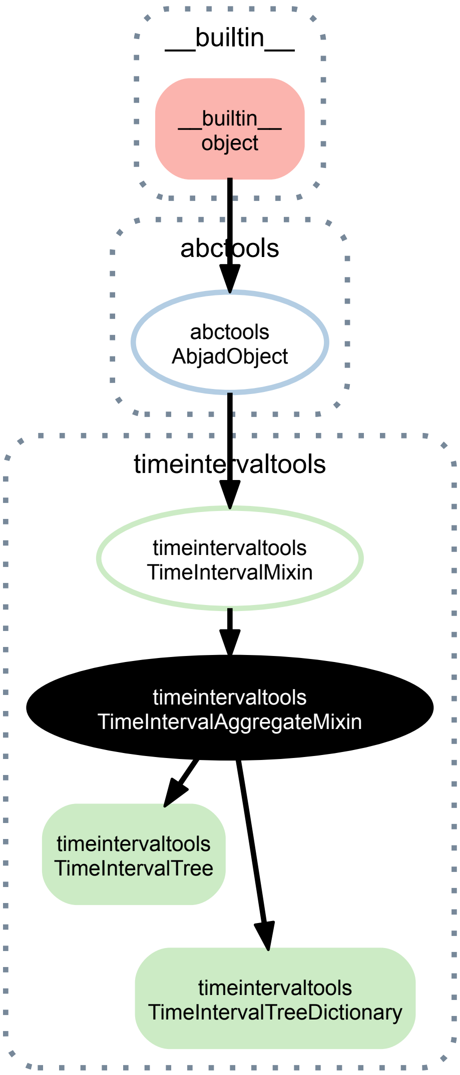
CHAPTER

THIRTYEIGHT

TIMEINTERVALTOOLS

38.1 Abstract Classes

38.1.1 timeintervaltools.TimeIntervalAggregateMixin



class `timeintervaltools.`**`TimeIntervalAggregateMixin`**(**args*, ***kwargs*)

Read-only properties

TimeIntervalAggregateMixin.**bounds**

Start and stop of self returned as TimeInterval instance:

```
>>> interval = TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

TimeIntervalAggregateMixin.**center**

Center offset of start and stop offsets:

```
>>> interval = TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

TimeIntervalAggregateMixin.**duration**

Duration of the time interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

TimeIntervalAggregateMixin.**earliest_start**

TimeIntervalAggregateMixin.**earliest_stop**

TimeIntervalAggregateMixin.**intervals**

TimeIntervalAggregateMixin.**latest_start**

TimeIntervalAggregateMixin.**latest_stop**

TimeIntervalAggregateMixin.**offset_counts**

TimeIntervalAggregateMixin.**offsets**

TimeIntervalAggregateMixin.**signature**

Tuple of start bound and stop bound.

```
>>> interval = TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

TimeIntervalAggregateMixin.**start**

Starting offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.start
Offset(2, 1)
```

Returns *Offset* instance.

TimeIntervalAggregateMixin.**stop**

Stopping offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.stop
Offset(10, 1)
```

Returns *Offset* instance.

TimeIntervalAggregateMixin.**storage_format**
Storage format of Abjad object.

Return string.

Methods

TimeIntervalAggregateMixin.**find_intervals_intersecting_or_tangent_to_interval**()

TimeIntervalAggregateMixin.**find_intervals_intersecting_or_tangent_to_offset**()

TimeIntervalAggregateMixin.**find_intervals_starting_after_offset**()

TimeIntervalAggregateMixin.**find_intervals_starting_and_stopping_within_interval**()

TimeIntervalAggregateMixin.**find_intervals_starting_at_offset**()

TimeIntervalAggregateMixin.**find_intervals_starting_before_offset**()

TimeIntervalAggregateMixin.**find_intervals_starting_or_stopping_at_offset**()

TimeIntervalAggregateMixin.**find_intervals_starting_within_interval**()

TimeIntervalAggregateMixin.**find_intervals_stopping_after_offset**()

TimeIntervalAggregateMixin.**find_intervals_stopping_at_offset**()

TimeIntervalAggregateMixin.**find_intervals_stopping_before_offset**()

TimeIntervalAggregateMixin.**find_intervals_stopping_within_interval**()

TimeIntervalAggregateMixin.**get_overlap_with_interval**(*interval*)
Return amount of overlap with *interval*.

TimeIntervalAggregateMixin.**is_contained_by_interval**(*interval*)
True if interval is contained by *interval*.

TimeIntervalAggregateMixin.**is_container_of_interval**(*interval*)
True if interval contains *interval*.

TimeIntervalAggregateMixin.**is_overlapped_by_interval**(*interval*)
True if interval is overlapped by *interval*.

TimeIntervalAggregateMixin.**is_tangent_to_interval**(*interval*)
True if interval is tangent to *interval*.

TimeIntervalAggregateMixin.**quantize_to_rational**(*rational*)

TimeIntervalAggregateMixin.**scale_by_rational**(*rational*)

TimeIntervalAggregateMixin.**scale_to_rational**(*rational*)

TimeIntervalAggregateMixin.**shift_by_rational**(*rational*)

TimeIntervalAggregateMixin.**shift_to_rational**(*rational*)

TimeIntervalAggregateMixin.**split_at_rationals**(**rationals*)

Special methods

TimeIntervalAggregateMixin.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

TimeIntervalAggregateMixin.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalAggregateMixin.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TimeIntervalAggregateMixin.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalAggregateMixin.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalAggregateMixin.**__ne__**(*expr*)
Defined equal to the opposite of equality.

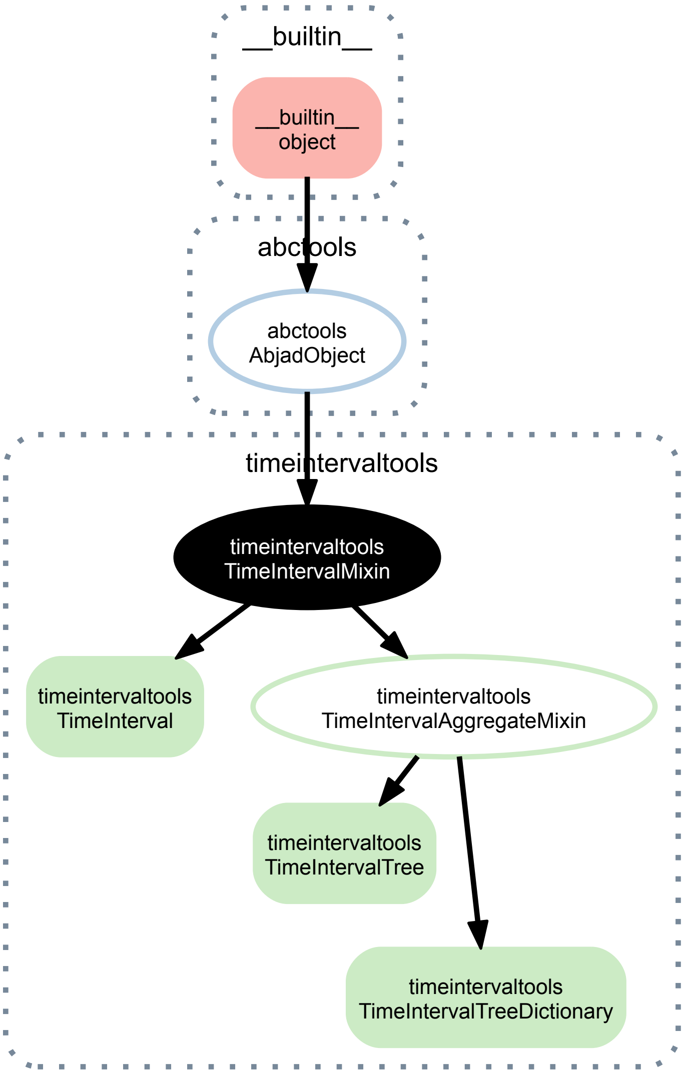
Return boolean.

TimeIntervalAggregateMixin.**__nonzero__**()

TimeIntervalAggregateMixin.**__repr__**()
Interpreter representation of Abjad object.

Return string.

38.1.2 timeintervaltools.TimeIntervalMixin



class `timeintervaltools.`**`TimeIntervalMixin`**(**args*, ***kwargs*)

Time-interval mixin.

For examples:

```
>>> from abjad.tools.timeintervaltools import TimeInterval
```

Time-interval mixins provide time-interval functionality.

Read-only properties

`TimeIntervalMixin.`**`bounds`**

Start and stop of self returned as TimeInterval instance:

```
>>> interval = TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`TimeIntervalMixin.`**`center`**
Center offset of start and stop offsets:

```
>>> interval = TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`TimeIntervalMixin.`**`duration`**
Duration of the time interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

`TimeIntervalMixin.`**`signature`**
Tuple of start bound and stop bound.

```
>>> interval = TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`TimeIntervalMixin.`**`start`**
Starting offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.start
Offset(2, 1)
```

Returns *Offset* instance.

`TimeIntervalMixin.`**`stop`**
Stopping offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.stop
Offset(10, 1)
```

Returns *Offset* instance.

`TimeIntervalMixin.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`TimeIntervalMixin.`**`get_overlap_with_interval`**(*interval*)
Return amount of overlap with *interval*.

`TimeIntervalMixin.`**`is_contained_by_interval`**(*interval*)
True if interval is contained by *interval*.

`TimeIntervalMixin.`**`is_container_of_interval`**(*interval*)
True if interval contains *interval*.

`TimeIntervalMixin.`**`is_overlapped_by_interval`**(*interval*)
True if interval is overlapped by *interval*.

`TimeIntervalMixin.`**`is_tangent_to_interval`**(*interval*)
True if interval is tangent to *interval*.

`TimeIntervalMixin.`**`quantize_to_rational`**(*rational*)

TimeIntervalMixin.**scale_by_rational**(*rational*)

TimeIntervalMixin.**scale_to_rational**(*rational*)

TimeIntervalMixin.**shift_by_rational**(*rational*)

TimeIntervalMixin.**shift_to_rational**(*rational*)

TimeIntervalMixin.**split_at_rationals**(**rationals*)

Special methods

TimeIntervalMixin.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

TimeIntervalMixin.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalMixin.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TimeIntervalMixin.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalMixin.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeIntervalMixin.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

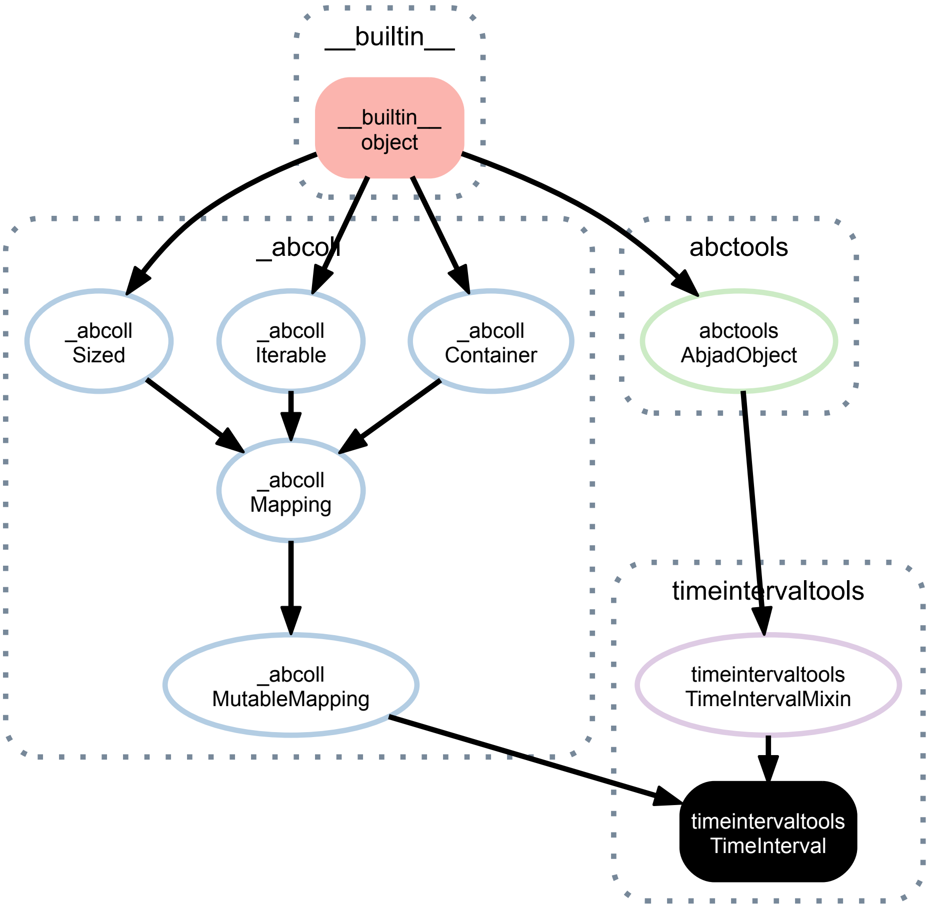
TimeIntervalMixin.**__nonzero__**()

TimeIntervalMixin.**__repr__**()
Interpreter representation of Abjad object.

Return string.

38.2 Concrete Classes

38.2.1 timeintervaltools.TimeInterval



class `timeintervaltools.`**`TimeInterval`**`(*args)`
A start / stop pair, carrying some metadata.

Read-only properties

`TimeInterval.`**`bounds`**
Start and stop of self returned as TimeInterval instance:

```
>>> interval = TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`TimeInterval.`**`center`**
Center point of start and stop bounds.

`TimeInterval.`**`duration`**
stop bound minus start bound.

`TimeInterval.`**`signature`**
Tuple of start bound and stop bound.

`TimeInterval.`**`start`**
start bound.

`TimeInterval.`**`stop`**
stop bound.

`TimeInterval.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`TimeInterval.`**`clear`**`()`

`TimeInterval.`**`get`**`(`*key*, *default=None*`)`

`TimeInterval.`**`get_overlap_with_interval`**`(`*interval*`)`
Return amount of overlap with *interval*.

`TimeInterval.`**`is_contained_by_interval`**`(`*interval*`)`
True if interval is contained by *interval*.

`TimeInterval.`**`is_container_of_interval`**`(`*interval*`)`
True if interval contains *interval*.

`TimeInterval.`**`is_overlapped_by_interval`**`(`*interval*`)`
True if interval is overlapped by *interval*.

`TimeInterval.`**`is_tangent_to_interval`**`(`*interval*`)`
True if interval is tangent to *interval*.

`TimeInterval.`**`items`**`()`

`TimeInterval.`**`iteritems`**`()`

`TimeInterval.`**`iterkeys`**`()`

`TimeInterval.`**`itervalues`**`()`

`TimeInterval.`**`keys`**`()`

`TimeInterval.`**`pop`**`(`*key*, *default=<object object at 0x1002b0040>*`)`

`TimeInterval.`**`popitem`**`()`

`TimeInterval.`**`quantize_to_rational`**`(`*rational*`)`

`TimeInterval.`**`scale_by_rational`**`(`*rational*`)`

`TimeInterval.`**`scale_to_rational`**`(`*rational*`)`

`TimeInterval.`**`setdefault`**`(`*key*, *default=None*`)`

`TimeInterval.`**`shift_by_rational`**`(`*rational*`)`

`TimeInterval.`**`shift_to_rational`**`(`*rational*`)`

`TimeInterval.`**`split_at_rationals`**`(`**rationals*`)`

`TimeInterval.`**`update`**`(`**args*, ***kwds*`)`

`TimeInterval.`**`values`**`()`

Special methods

`TimeInterval.`**`__contains__`**`(`*key*`)`

`TimeInterval.`**`__delitem__`**`(`*item*`)`

TimeInterval.**__eq__**(*expr*)

TimeInterval.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeInterval.**__getitem__**(*item*)

TimeInterval.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TimeInterval.**__hash__**()

TimeInterval.**__iter__**()

TimeInterval.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimeInterval.**__len__**()

TimeInterval.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

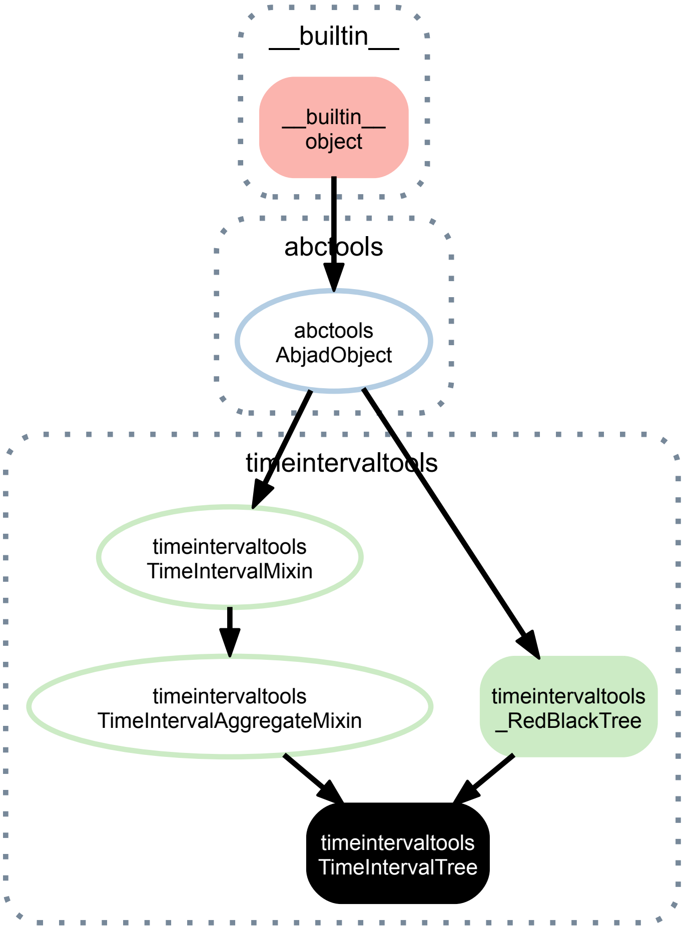
TimeInterval.**__ne__**(*expr*)

TimeInterval.**__nonzero__**()

TimeInterval.**__repr__**()

TimeInterval.**__setitem__**(*item*, *value*)

38.2.2 timeintervaltools.TimeIntervalTree



class `timeintervaltools.`**`TimeIntervalTree`**(*intervals=None*)

An augmented red-black tree for storing and searching for intervals of time (rather than pitch).

This allows for the arbitrary placement of blocks of material along a time-line. While this functionality could be achieved with Python's built-in collections, this class reduces the complexity of the search process, such as locating overlapping intervals.

TimeIntervalTrees can be instantiated without contents, or from a mixed collection of other TimeIntervalTrees and / or TimeIntervals. The input will be parsed recursively:

```
>>> from abjad.tools.timeintervaltools import *
```

```
>>> interval_one = TimeInterval(0, 10)
>>> interval_two = TimeInterval(1, 8)
>>> interval_three = TimeInterval(3, 13)
>>> tree = TimeIntervalTree([interval_one, interval_two, interval_three])
```

```
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(10, 1), {}),
    TimeInterval(Offset(1, 1), Offset(8, 1), {}),
    TimeInterval(Offset(3, 1), Offset(13, 1), {})
])
```

Return *TimeIntervalTree* instance.

Read-only properties

`TimeIntervalTree.`**`bounds`**

Start and stop of self returned as TimeInterval instance:

```
>>> interval = TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`TimeIntervalTree.`**`center`**

Center offset of start and stop offsets:

```
>>> interval = TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`TimeIntervalTree.`**`duration`**

Absolute difference of the stop and start values of the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.duration
Duration(5, 2)
```

Empty trees have a duration of 0.

Return `Duration` instance.

`TimeIntervalTree.`**`earliest_start`**

The minimum start value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.earliest_start
Offset(1, 1)
```

Return `Offset` instance, or None if tree is empty.

`TimeIntervalTree.`**`earliest_stop`**

The minimum stop value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.earliest_stop
Offset(2, 1)
```

Return `Offset` instance, or None if tree is empty.

`TimeIntervalTree.`**`intervals`**

`TimeIntervalTree.`**`latest_start`**

The maximum start value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.latest_start
Offset(3, 1)
```

Return `Offset` instance, or None if tree is empty.

`TimeIntervalTree.`**`latest_stop`**

The maximum stop value of all intervals in the tree:

```
>>> ti1 = TimeInterval(1, 2)
>>> ti2 = TimeInterval(3, (7, 2))
>>> tree = TimeIntervalTree([ti1, ti2])
>>> tree.latest_stop
Offset(7, 2)
```

Return `Offset` instance, or None if tree is empty.

`TimeIntervalTree.`**`offset_counts`**

`TimeIntervalTree.`**`offsets`**

`TimeIntervalTree.`**`signature`**

Tuple of start bound and stop bound.

```
>>> interval = TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`TimeIntervalTree.`**`start`**

Starting offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.start
Offset(2, 1)
```

Returns *Offset* instance.

`TimeIntervalTree.`**`stop`**

Stopping offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.stop
Offset(10, 1)
```

Returns *Offset* instance.

`TimeIntervalTree.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`TimeIntervalTree.`**`find_intervals_intersecting_or_tangent_to_interval`**`(*args)`

Find all intervals in tree intersecting or tangent to the interval defined in *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval(0, 1)
>>> found = tree.find_intervals_intersecting_or_tangent_to_interval(interval)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

```
>>> interval = TimeInterval(3, 4)
>>> found = tree.find_intervals_intersecting_or_tangent_to_interval(interval)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_intersecting_or_tangent_to_offset**(*offset*)
Find all intervals in tree intersecting or tangent to *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = tree.find_intervals_intersecting_or_tangent_to_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

```
>>> offset = 3
>>> found = tree.find_intervals_intersecting_or_tangent_to_offset(offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_after_offset**(*offset*)
Find all intervals in tree starting after *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 0
>>> found = tree.find_intervals_starting_after_offset(offset)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_after_offset(offset)
>>> sorted([x['name'] for x in found])
['d']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_and_stopping_within_interval**(**args*)
Find all intervals in tree starting and stopping within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval(1, 3)
>>> found = tree.find_intervals_starting_and_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> interval = TimeInterval(-1, 2)
>>> found = tree.find_intervals_starting_and_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_at_offset**(*offset*)
Find all intervals in tree starting at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
```

```
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 0
>>> found = tree.find_intervals_starting_at_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_at_offset(offset)
>>> sorted([x['name'] for x in found])
['b']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_before_offset**(*offset*)
Find all intervals in tree starting before *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> offset = 2
>>> found = tree.find_intervals_starting_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_or_stopping_at_offset**(*offset*)
Find all intervals in tree starting or stopping at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 2
>>> found = tree.find_intervals_starting_or_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

```
>>> offset = 1
>>> found = tree.find_intervals_starting_or_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

Return *TimeIntervalTree* instance.

TimeIntervalTree.**find_intervals_starting_within_interval**(**args*)
Find all intervals in tree starting within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval((-1, 2), (1, 2))
>>> found = tree.find_intervals_starting_within_interval(interval)
```

```
>>> sorted([x['name'] for x in found])
['a', 'c']
```

```
>>> interval = TimeInterval((1, 2), (5, 2))
>>> found = tree.find_intervals_starting_within_interval(interval)
>>> sorted([x['name'] for x in found])
['b', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`find_intervals_stopping_after_offset`**(*offset*)

Find all intervals in tree stopping after *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 1
>>> found = tree.find_intervals_stopping_after_offset(offset)
>>> sorted([x['name'] for x in found])
['b', 'c', 'd']
```

```
>>> offset = 2
>>> found = tree.find_intervals_stopping_after_offset(offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`find_intervals_stopping_at_offset`**(*offset*)

Find all intervals in tree stopping at *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 3
>>> found = tree.find_intervals_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

```
>>> offset = 1
>>> found = tree.find_intervals_stopping_at_offset(offset)
>>> sorted([x['name'] for x in found])
['a']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`find_intervals_stopping_before_offset`**(*offset*)

Find all intervals in tree stopping before *offset*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> offset = 3
>>> found = tree.find_intervals_stopping_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b']
```

```
>>> offset = (7, 2)
>>> found = tree.find_intervals_stopping_before_offset(offset)
>>> sorted([x['name'] for x in found])
['a', 'b', 'c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`find_intervals_stopping_within_interval`**`(`**args*`)`

Find all intervals in tree stopping within the interval defined by *args*:

```
>>> a = TimeInterval(0, 1, {'name': 'a'})
>>> b = TimeInterval(1, 2, {'name': 'b'})
>>> c = TimeInterval(0, 3, {'name': 'c'})
>>> d = TimeInterval(2, 3, {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
```

```
>>> interval = TimeInterval((3, 2), (5, 2))
>>> found = tree.find_intervals_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['b']
```

```
>>> interval = TimeInterval((5, 2), (7, 2))
>>> found = tree.find_intervals_stopping_within_interval(interval)
>>> sorted([x['name'] for x in found])
['c', 'd']
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`get_overlap_with_interval`**`(`*interval*`)`

Return amount of overlap with *interval*.

`TimeIntervalTree.`**`is_contained_by_interval`**`(`*interval*`)`

True if interval is contained by *interval*.

`TimeIntervalTree.`**`is_container_of_interval`**`(`*interval*`)`

True if interval contains *interval*.

`TimeIntervalTree.`**`is_overlapped_by_interval`**`(`*interval*`)`

True if interval is overlapped by *interval*.

`TimeIntervalTree.`**`is_tangent_to_interval`**`(`*interval*`)`

True if interval is tangent to *interval*.

`TimeIntervalTree.`**`quantize_to_rational`**`(`*rational*`)`

Quantize all intervals in tree to a multiple (1 or more) of *rational*:

```
>>> a = TimeInterval((1, 16), (1, 8), {'name': 'a'})
>>> b = TimeInterval((2, 7), (13, 7), {'name': 'b'})
>>> c = TimeInterval((3, 5), (8, 5), {'name': 'c'})
>>> d = TimeInterval((2, 3), (5, 3), {'name': 'd'})
>>> tree = TimeIntervalTree([a, b, c, d])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(1, 16), Offset(1, 8), {'name': 'a'}),
    TimeInterval(Offset(2, 7), Offset(13, 7), {'name': 'b'}),
    TimeInterval(Offset(3, 5), Offset(8, 5), {'name': 'c'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'})
])
```

```
>>> rational = (1, 4)
>>> tree.quantize_to_rational(rational)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'a'}),
    TimeInterval(Offset(1, 4), Offset(7, 4), {'name': 'b'}),
    TimeInterval(Offset(1, 2), Offset(3, 2), {'name': 'c'}),
    TimeInterval(Offset(3, 4), Offset(7, 4), {'name': 'd'})
])
```

```
>>> rational = (1, 3)
>>> tree.quantize_to_rational(rational)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 3), {'name': 'a'}),
    TimeInterval(Offset(1, 3), Offset(2, 1), {'name': 'b'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'c'}),
    TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`scale_by_rational`**(*rational*)

Scale aggregate duration of tree by *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.scale_by_rational((2, 3))
>>> result
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(2, 3), {'name': 'one'}),
    TimeInterval(Offset(1, 3), Offset(5, 3), {'name': 'two'}),
    TimeInterval(Offset(4, 3), Offset(8, 3), {'name': 'three'})
])
```

Scaling works regardless of the starting offset of the *TimeIntervalTree*:

```
>>> zero = TimeInterval(-4, 0, {'name': 'zero'})
>>> tree = TimeIntervalTree([zero, one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.scale_by_rational(2)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(4, 1), {'name': 'zero'}),
    TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'one'}),
    TimeInterval(Offset(5, 1), Offset(9, 1), {'name': 'two'}),
    TimeInterval(Offset(8, 1), Offset(12, 1), {'name': 'three'})
])
```

```
>>> result.start == tree.start
True
>>> result.duration == tree.duration * 2
True
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`scale_to_rational`**(*rational*)

Scale aggregate duration of tree to *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.scale_to_rational(1)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'one'}),
    TimeInterval(Offset(1, 8), Offset(5, 8), {'name': 'two'}),
```

```
        TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'three'})
    ])
```

```
>>> result.scale_to_rational(10)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(5, 2), {'name': 'one'}),
    TimeInterval(Offset(5, 4), Offset(25, 4), {'name': 'two'}),
    TimeInterval(Offset(5, 1), Offset(10, 1), {'name': 'three'})
])
```

Scaling works regardless of the starting offset of the *TimeIntervalTree*:

```
>>> zero = TimeInterval(-4, 0, {'name': 'zero'})
>>> tree = TimeIntervalTree([zero, one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> tree.scale_to_rational(4)
TimeIntervalTree([
    TimeInterval(Offset(-4, 1), Offset(-2, 1), {'name': 'zero'}),
    TimeInterval(Offset(-2, 1), Offset(-3, 2), {'name': 'one'}),
    TimeInterval(Offset(-7, 4), Offset(-3, 4), {'name': 'two'}),
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`shift_by_rational`**(*rational*)

Shift aggregate offset of tree by *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.shift_by_rational(-2.5)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(-5, 2), Offset(-3, 2), {'name': 'one'}),
    TimeInterval(Offset(-2, 1), Offset(0, 1), {'name': 'two'}),
    TimeInterval(Offset(-1, 2), Offset(3, 2), {'name': 'three'})
])
>>> result.shift_by_rational(6)
TimeIntervalTree([
    TimeInterval(Offset(7, 2), Offset(9, 2), {'name': 'one'}),
    TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'two'}),
    TimeInterval(Offset(11, 2), Offset(15, 2), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`shift_to_rational`**(*rational*)

Shift aggregate offset of tree to *rational*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
```

```
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
    ])
```

```
>>> result = tree.shift_to_rational(100)
>>> result
TimeIntervalTree([
    TimeInterval(Offset(100, 1), Offset(101, 1), {'name': 'one'}),
    TimeInterval(Offset(201, 2), Offset(205, 2), {'name': 'two'}),
    TimeInterval(Offset(102, 1), Offset(104, 1), {'name': 'three'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTree.`**`split_at_rationals`**`(*rationals)`
Split tree at each rational in *rationals*:

```
>>> one = TimeInterval(0, 1, {'name': 'one'})
>>> two = TimeInterval((1, 2), (5, 2), {'name': 'two'})
>>> three = TimeInterval(2, 4, {'name': 'three'})
>>> tree = TimeIntervalTree([one, two, three])
>>> tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'})
])
```

```
>>> result = tree.split_at_rationals(1, 2, 3)
>>> len(result)
4
```

```
>>> result[0]
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'two'})
])
```

```
>>> result[1]
TimeIntervalTree([
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'two'})
])
```

```
>>> result[2]
TimeIntervalTree([
    TimeInterval(Offset(2, 1), Offset(5, 2), {'name': 'two'}),
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'three'})
])
```

```
>>> result[3]
TimeIntervalTree([
    TimeInterval(Offset(3, 1), Offset(4, 1), {'name': 'three'})
])
```

Return tuple of *TimeIntervalTree* instances.

Special methods

`TimeIntervalTree.`**`__contains__`**`(item)`

`TimeIntervalTree.`**`__copy__`**`()`

`TimeIntervalTree.`**`__eq__`**`(expr)`

`TimeIntervalTree.`**`__ge__`**`(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TimeIntervalTree.`**`__getitem__`**`(`*item*`)`

`TimeIntervalTree.`**`__getslice__`**`(`*start*, *end*`)`

`TimeIntervalTree.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`TimeIntervalTree.`**`__iter__`**`()`

`TimeIntervalTree.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`TimeIntervalTree.`**`__len__`**`()`

`TimeIntervalTree.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`TimeIntervalTree.`**`__ne__`**`(`*expr*`)`

`TimeIntervalTree.`**`__nonzero__`**`()`

TimeIntervalTree evaluates to True if it contains any intervals:

```
>>> true_tree = TimeIntervalTree([TimeInterval(0, 1)])
>>> false_tree = TimeIntervalTree([])
```

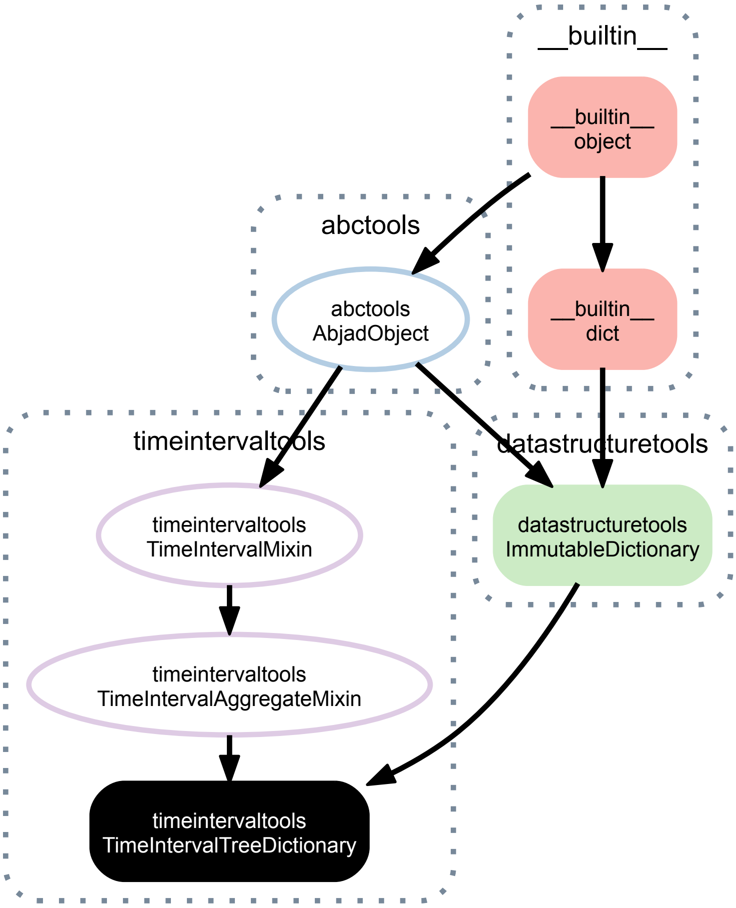
```
>>> bool(true_tree)
True
>>> bool(false_tree)
False
```

Return boolean.

`TimeIntervalTree.`**`__repr__`**`()`

`TimeIntervalTree.`**`__str__`**`()`

38.2.3 timeintervaltools.TimeIntervalTreeDictionary



class `timeintervaltools.`**`TimeIntervalTreeDictionary`**`(*args)`

A dictionary of *TimeIntervalTrees*:

```
>>> from abjad.tools.timeintervaltools import TimeIntervalTreeDictionary
```

```
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
>>> from abjad.tools.timeintervaltools import TimeInterval
```

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

TimeIntervalTreeDictionary can be instantiated from one or more other *TimeIntervalTreeDictionary* instances, whose trees will be fused if they share keys. It can also be instantiated from a regular dictionary whose values are *TimeIntervalTree* instances, or from a list of pairs where the second value of each pair is a

TimeIntervalTree instance.

TimeIntervalTreeDictionary supports the same set of methods and properties as *TimeIntervalTree* and *TimeInterval*, including searching for intervals, quantizing, scaling, shifting and splitting.

TimeIntervalTreeDictionary is immutable.

Return *TimeIntervalTreeDictionary* instance.

Read-only properties

`TimeIntervalTreeDictionary.`**`bounds`**

Start and stop of self returned as TimeInterval instance:

```
>>> interval = TimeInterval(2, 10)
>>> bounds = interval.bounds
>>> bounds
TimeInterval(Offset(2, 1), Offset(10, 1), {})
>>> bounds == interval
True
>>> bounds is interval
False
```

Returns *TimeInterval* instance.

`TimeIntervalTreeDictionary.`**`center`**

Center offset of start and stop offsets:

```
>>> interval = TimeInterval(2, 10)
>>> interval.center
Offset(6, 1)
```

Returns *Offset* instance.

`TimeIntervalTreeDictionary.`**`composite_tree`**

The *TimeIntervalTree* composed of all the intervals in all trees in self:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.composite_tree
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'})
])
```

Return *TimeIntervalTree* instance.

`TimeIntervalTreeDictionary.`**`duration`**

Duration of the time interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.duration
Duration(8, 1)
```

Returns *Duration* instance.

`TimeIntervalTreeDictionary.`**`earliest_start`**

The earliest start offset of all intervals in all trees in self:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.earliest_start
Offset(0, 1)
```

Return *Offset* instance.

`TimeIntervalTreeDictionary.`**`earliest_stop`**

The earliest stop offset of all intervals in all trees in self:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.earliest_stop
Offset(1, 1)
```

Return *Offset* instance.

`TimeIntervalTreeDictionary.`**`intervals`**

`TimeIntervalTreeDictionary.`**`latest_start`**

The latest start offset of all intervals in all trees in self:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.latest_start
Offset(2, 1)
```

Return *Offset* instance.

`TimeIntervalTreeDictionary.`**`latest_stop`**

The latest stop offset of all intervals in all trees in self:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
```

```
>>> treedict.latest_stop
Offset(3, 1)
```

Return *Offset* instance.

`TimeIntervalTreeDictionary.`**`offset_counts`**

`TimeIntervalTreeDictionary.`**`offsets`**

`TimeIntervalTreeDictionary.`**`signature`**

Tuple of start bound and stop bound.

```
>>> interval = TimeInterval(2, 10)
>>> interval.signature
(Offset(2, 1), Offset(10, 1))
```

Returns 2-tuple of *Offset* instances.

`TimeIntervalTreeDictionary.`**`start`**

Starting offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.start
Offset(2, 1)
```

Returns *Offset* instance.

TimeIntervalTreeDictionary.**stop**
Stopping offset of interval:

```
>>> interval = TimeInterval(2, 10)
>>> interval.stop
Offset(10, 1)
```

Returns *Offset* instance.

TimeIntervalTreeDictionary.**storage_format**
Storage format of Abjad object.

Return string.

Methods

TimeIntervalTreeDictionary.**clear**() → None. Remove all items from D.

TimeIntervalTreeDictionary.**copy**() → a shallow copy of D

TimeIntervalTreeDictionary.**find_intervals_intersecting_or_tangent_to_interval**(**args*)
Find all intervals in dictionary intersecting or tangent to the interval defined in *args*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> interval = TimeInterval(0, 1)
>>> treedict.find_intervals_intersecting_or_tangent_to_interval(interval)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

```
>>> interval = TimeInterval(3, 4)
>>> treedict.find_intervals_intersecting_or_tangent_to_interval(interval)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_intersecting_or_tangent_to_offset**(*offset*)

Find all intervals in dictionary intersecting or tangent to *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_intersecting_or_tangent_to_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

```
>>> offset = 3
>>> treedict.find_intervals_intersecting_or_tangent_to_offset(offset)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_starting_after_offset**(*offset*)

Find all intervals in dictionary starting after *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
```

```
        ]),
    })
```

```
    >>> offset = 0
    >>> treedict.find_intervals_starting_after_offset(offset)
    TimeIntervalTreeDictionary({
        'b': TimeIntervalTree([
            TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
        ]),
        'd': TimeIntervalTree([
            TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
        ]),
    })
```

```
    >>> offset = 1
    >>> treedict.find_intervals_starting_after_offset(offset)
    TimeIntervalTreeDictionary({
        'd': TimeIntervalTree([
            TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
        ]),
    })
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_starting_and_stopping_within_interval**(**args*)

Find all intervals in dictionary starting and stopping within the interval defined by *args*:

```
    >>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
    >>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
    >>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
    >>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
    >>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
    >>> treedict
    TimeIntervalTreeDictionary({
        'a': TimeIntervalTree([
            TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
        ]),
        'b': TimeIntervalTree([
            TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
        ]),
        'c': TimeIntervalTree([
            TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
        ]),
        'd': TimeIntervalTree([
            TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
        ]),
    })
```

```
    >>> interval = TimeInterval(1, 3)
    >>> treedict.find_intervals_starting_and_stopping_within_interval(interval)
    TimeIntervalTreeDictionary({
        'b': TimeIntervalTree([
            TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
        ]),
        'd': TimeIntervalTree([
            TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
        ]),
    })
```

```
    >>> interval = TimeInterval(-1, 2)
    >>> treedict.find_intervals_starting_and_stopping_within_interval(interval)
    TimeIntervalTreeDictionary({
        'a': TimeIntervalTree([
            TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
        ]),
        'b': TimeIntervalTree([
            TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
        ]),
    })
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_starting_at_offset**(*offset*)

Find all intervals in dictionary starting at *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 0
>>> treedict.find_intervals_starting_at_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_starting_at_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_starting_before_offset**(*offset*)

Find all intervals in dictionary starting before *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_starting_before_offset(offset)
TimeIntervalTreeDictionary({
```

```
        'a': TimeIntervalTree([
            TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
        ]),
        'c': TimeIntervalTree([
            TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
        ]),
    })
```

```
>>> offset = 2
>>> treedict.find_intervals_starting_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_starting_or_stopping_at_offset**(*offset*)

Find all intervals in dictionary starting or stopping at *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 2
>>> treedict.find_intervals_starting_or_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_starting_or_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.`**`find_intervals_starting_within_interval`**`(*args)`

Find all intervals in dictionary starting within the interval defined by *args*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> interval = TimeInterval((-1, 2), (1, 2))
>>> treedict.find_intervals_starting_within_interval(interval)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
})
```

```
>>> interval = TimeInterval((1, 2), (5, 2))
>>> treedict.find_intervals_starting_within_interval(interval)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.`**`find_intervals_stopping_after_offset`**`(offset)`

Find all intervals in dictionary stopping after *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_stopping_after_offset(offset)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 2
>>> treedict.find_intervals_stopping_after_offset(offset)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_stopping_at_offset**(*offset*)

Find all intervals in dictionary stopping at *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 3
>>> treedict.find_intervals_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 1
>>> treedict.find_intervals_stopping_at_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_stopping_before_offset**(*offset*)
Find all intervals in dictionary stopping before *offset*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

```
>>> offset = 3
>>> treedict.find_intervals_stopping_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})
```

```
>>> offset = (7, 2)
>>> treedict.find_intervals_stopping_before_offset(offset)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**find_intervals_stopping_within_interval**(**args*)
Find all intervals in dictionary stopping within the interval defined by *args*:

```
>>> a = TimeIntervalTree([TimeInterval(0, 1, {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval(1, 2, {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval(0, 3, {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval(2, 3, {'name': 'd'})])
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
```

```
        ]),
        'd': TimeIntervalTree([
            TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
        ]),
    })
```

```
>>> interval = TimeInterval((3, 2), (5, 2))
>>> treedict.find_intervals_stopping_within_interval(interval)
TimeIntervalTreeDictionary({
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'b'}),
    ]),
})
```

```
>>> interval = TimeInterval((5, 2), (7, 2))
>>> treedict.find_intervals_stopping_within_interval(interval)
TimeIntervalTreeDictionary({
    'c': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(3, 1), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.`**`get`**(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`TimeIntervalTreeDictionary.`**`get_overlap_with_interval`**(*interval*)
Return amount of overlap with *interval*.

`TimeIntervalTreeDictionary.`**`has_key`**(*k*) → True if D has a key k, else False

`TimeIntervalTreeDictionary.`**`is_contained_by_interval`**(*interval*)
True if interval is contained by *interval*.

`TimeIntervalTreeDictionary.`**`is_container_of_interval`**(*interval*)
True if interval contains *interval*.

`TimeIntervalTreeDictionary.`**`is_overlapped_by_interval`**(*interval*)
True if interval is overlapped by *interval*.

`TimeIntervalTreeDictionary.`**`is_tangent_to_interval`**(*interval*)
True if interval is tangent to *interval*.

`TimeIntervalTreeDictionary.`**`items`**() → list of D's (key, value) pairs, as 2-tuples

`TimeIntervalTreeDictionary.`**`iteritems`**() → an iterator over the (key, value) items of D

`TimeIntervalTreeDictionary.`**`iterkeys`**() → an iterator over the keys of D

`TimeIntervalTreeDictionary.`**`itervalues`**() → an iterator over the values of D

`TimeIntervalTreeDictionary.`**`keys`**() → list of D's keys

`TimeIntervalTreeDictionary.`**`pop`**(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`TimeIntervalTreeDictionary.`**`popitem`**() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

`TimeIntervalTreeDictionary.`**`quantize_to_rational`**(*rational*)
Quantize all intervals in dictionary to a multiple (1 or more) of *rational*:

```
>>> a = TimeIntervalTree([TimeInterval((1, 16), (1, 8), {'name': 'a'})])
>>> b = TimeIntervalTree([TimeInterval((2, 7), (13, 7), {'name': 'b'})])
>>> c = TimeIntervalTree([TimeInterval((3, 5), (8, 5), {'name': 'c'})])
>>> d = TimeIntervalTree([TimeInterval((2, 3), (5, 3), {'name': 'd'})])
```

```
>>> treedict = TimeIntervalTreeDictionary({'a': a, 'b': b, 'c': c, 'd': d})
>>> treedict
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(1, 16), Offset(1, 8), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(2, 7), Offset(13, 7), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(3, 5), Offset(8, 5), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'}),
    ]),
})
```

```
>>> rational = (1, 4)
>>> treedict.quantize_to_rational(rational)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 4), Offset(7, 4), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(3, 2), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(3, 4), Offset(7, 4), {'name': 'd'}),
    ]),
})
```

```
>>> rational = (1, 3)
>>> treedict.quantize_to_rational(rational)
TimeIntervalTreeDictionary({
    'a': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 3), {'name': 'a'}),
    ]),
    'b': TimeIntervalTree([
        TimeInterval(Offset(1, 3), Offset(2, 1), {'name': 'b'}),
    ]),
    'c': TimeIntervalTree([
        TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'c'}),
    ]),
    'd': TimeIntervalTree([
        TimeInterval(Offset(2, 3), Offset(5, 3), {'name': 'd'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.`**`scale_by_rational`**(*rational*)

Scale aggregate duration of dictionary by *rational*:

```
>>> one = TimeIntervalTree([TimeInterval(0, 1, {'name': 'one'})])
>>> two = TimeIntervalTree([TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = TimeIntervalTree([TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = TimeIntervalTreeDictionary(
...     {'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
```

```
    })
```

```
>>> result = treedict.scale_by_rational((2, 3))
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(2, 3), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(4, 3), Offset(8, 3), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 3), Offset(5, 3), {'name': 'two'}),
    ]),
})
```

Scaling works regardless of the starting offset of the *TimeIntervalTreeDictionary*:

```
>>> zero = TimeIntervalTree([TimeInterval(-4, 0, {'name': 'zero'})])
>>> treedict = TimeIntervalTreeDictionary(
...     {'zero': zero, 'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(0, 1), {'name': 'zero'}),
    ]),
})
```

```
>>> result = treedict.scale_by_rational(2)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(8, 1), Offset(12, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(5, 1), Offset(9, 1), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(4, 1), {'name': 'zero'}),
    ]),
})
```

```
>>> result.start == treedict.start
True
>>> result.duration == treedict.duration * 2
True
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**scale_to_rational**(*rational*)

Scale aggregate duration of dictionary to *rational*:

```
>>> one = TimeIntervalTree([TimeInterval(0, 1, {'name': 'one'})])
>>> two = TimeIntervalTree([TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = TimeIntervalTree([TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = TimeIntervalTreeDictionary({'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
```

```
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.scale_to_rational(1)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 4), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 8), Offset(5, 8), {'name': 'two'}),
    ]),
})
```

```
>>> result.scale_to_rational(10)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(5, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(5, 1), Offset(10, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(5, 4), Offset(25, 4), {'name': 'two'}),
    ]),
})
```

Scaling works regardless of the starting offset of the *TimeIntervalTreeDictionary*:

```
>>> zero = TimeIntervalTree([TimeInterval(-4, 0, {'name': 'zero'})])
>>> treedict = TimeIntervalTreeDictionary(
...     {'zero': zero, 'one': one, 'two': two, 'three': three})
```

```
>>> treedict.scale_to_rational(4)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(-2, 1), Offset(-3, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(-1, 1), Offset(0, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(-7, 4), Offset(-3, 4), {'name': 'two'}),
    ]),
    'zero': TimeIntervalTree([
        TimeInterval(Offset(-4, 1), Offset(-2, 1), {'name': 'zero'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

`TimeIntervalTreeDictionary.`**`setdefault`**(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

`TimeIntervalTreeDictionary.`**`shift_by_rational`**(*rational*)

Shift aggregate offset of dictionary by *rational*:

```
>>> one = TimeIntervalTree([TimeInterval(0, 1, {'name': 'one'})])
>>> two = TimeIntervalTree([TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = TimeIntervalTree([TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = TimeIntervalTreeDictionary({'one': one, 'two': two, 'three': three})
>>> treedict
```

```
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.shift_by_rational(-2.5)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(-5, 2), Offset(-3, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(-1, 2), Offset(3, 2), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(-2, 1), Offset(0, 1), {'name': 'two'}),
    ]),
})
```

```
>>> result.shift_by_rational(6)
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(7, 2), Offset(9, 2), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(11, 2), Offset(15, 2), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(4, 1), Offset(6, 1), {'name': 'two'}),
    ]),
})
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**shift_to_rational**(*rational*)

Shift aggregate offset of dictionary to *rational*:

```
>>> one = TimeIntervalTree([TimeInterval(0, 1, {'name': 'one'})])
>>> two = TimeIntervalTree([TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = TimeIntervalTree([TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = TimeIntervalTreeDictionary({'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.shift_to_rational(100)
>>> result
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(100, 1), Offset(101, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(102, 1), Offset(104, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(201, 2), Offset(205, 2), {'name': 'two'}),
```

```
        ]),
    })
```

Return *TimeIntervalTreeDictionary* instance.

TimeIntervalTreeDictionary.**split_at_rationals**(**rationals*)
Split dictionary at each rational in *rationals*:

```
>>> one = TimeIntervalTree([TimeInterval(0, 1, {'name': 'one'})])
>>> two = TimeIntervalTree([TimeInterval((1, 2), (5, 2), {'name': 'two'})])
>>> three = TimeIntervalTree([TimeInterval(2, 4, {'name': 'three'})])
>>> treedict = TimeIntervalTreeDictionary({'one': one, 'two': two, 'three': three})
>>> treedict
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(4, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result = treedict.split_at_rationals(1, 2, 3)
>>> len(result)
4
```

```
>>> result[0]
TimeIntervalTreeDictionary({
    'one': TimeIntervalTree([
        TimeInterval(Offset(0, 1), Offset(1, 1), {'name': 'one'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 2), Offset(1, 1), {'name': 'two'}),
    ]),
})
```

```
>>> result[1]
TimeIntervalTreeDictionary({
    'two': TimeIntervalTree([
        TimeInterval(Offset(1, 1), Offset(2, 1), {'name': 'two'}),
    ]),
})
```

```
>>> result[2]
TimeIntervalTreeDictionary({
    'three': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(3, 1), {'name': 'three'}),
    ]),
    'two': TimeIntervalTree([
        TimeInterval(Offset(2, 1), Offset(5, 2), {'name': 'two'}),
    ]),
})
```

```
>>> result[3]
TimeIntervalTreeDictionary({
    'three': TimeIntervalTree([
        TimeInterval(Offset(3, 1), Offset(4, 1), {'name': 'three'}),
    ]),
})
```

Return tuple of *TimeIntervalTreeDictionary* instances.

TimeIntervalTreeDictionary.**update**([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

TimeIntervalTreeDictionary.**values**() → list of D's values

`TimeIntervalTreeDictionary.`**`viewitems`**`()` → a set-like object providing a view on D's items

`TimeIntervalTreeDictionary.`**`viewkeys`**`()` → a set-like object providing a view on D's keys

`TimeIntervalTreeDictionary.`**`viewvalues`**`()` → an object providing a view on D's values

Special methods

`TimeIntervalTreeDictionary.`**`__cmp__`**`(`*y*`)` *<==> cmp(x, y)*

`TimeIntervalTreeDictionary.`**`__contains__`**`(`*k*`)` → True if D has a key k, else False

`TimeIntervalTreeDictionary.`**`__delitem__`**`(`**args*`)`

`TimeIntervalTreeDictionary.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`TimeIntervalTreeDictionary.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`TimeIntervalTreeDictionary.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`TimeIntervalTreeDictionary.`**`__gt__`**`()`
x.__gt__(y) <==> x>y

`TimeIntervalTreeDictionary.`**`__iter__`**`()` *<==> iter(x)*

`TimeIntervalTreeDictionary.`**`__le__`**`()`
x.__le__(y) <==> x<=y

`TimeIntervalTreeDictionary.`**`__len__`**`()` *<==> len(x)*

`TimeIntervalTreeDictionary.`**`__lt__`**`()`
x.__lt__(y) <==> x<y

`TimeIntervalTreeDictionary.`**`__ne__`**`()`
x.__ne__(y) <==> x!=y

`TimeIntervalTreeDictionary.`**`__nonzero__`**`()`

`TimeIntervalTreeDictionary.`**`__repr__`**`()`

`TimeIntervalTreeDictionary.`**`__setitem__`**`(`**args*`)`

38.3 Functions

38.3.1 timeintervaltools.all_are_intervals_or_trees_or_empty

`timeintervaltools.`**`all_are_intervals_or_trees_or_empty`**`(`*input*`)`
Recursively test if all elements of *input* are TimeIntervals or TimeIntervalTrees. An empty result also return as True.

38.3.2 timeintervaltools.all_intervals_are_contiguous

`timeintervaltools.`**`all_intervals_are_contiguous`**`(`*intervals*`)`
True when all intervals in *intervals* are contiguous and non-overlapping.

38.3.3 timeintervaltools.all_intervals_are_nonoverlapping

`timeintervaltools.`**`all_intervals_are_nonoverlapping`**`(`*intervals*`)`
True when all intervals in *intervals* in tree are non-overlapping.

38.3.4 timeintervaltools.calculate_density_of_attacks_in_interval

`timeintervaltools.calculate_density_of_attacks_in_interval`(*intervals, interval*)

Calculate the number of attacks in *interval* over the duration of *interval*.

Return Fraction.

38.3.5 timeintervaltools.calculate_density_of_releases_in_interval

`timeintervaltools.calculate_density_of_releases_in_interval`(*intervals, interval*)

Calculate the number of releases in *interval* divided by the duration of *interval*.

Return Fraction.

38.3.6 timeintervaltools.calculate_depth_centroid_of_intervals

`timeintervaltools.calculate_depth_centroid_of_intervals`(*intervals*)

Calculate the weighted mean offset of *intervals*, such that the centroids of each interval in the depth tree of *intervals* make up the values of the mean, and the depth of each interval in the depth tree of *intervals* make up the weights.

Return Offset.

38.3.7 timeintervaltools.calculate_depth_centroid_of_intervals_in_interval

`timeintervaltools.calculate_depth_centroid_of_intervals_in_interval`(*intervals, interval*)

Return the weighted mean of the depth tree of *intervals* in *interval*, such that the centroids of each interval of the depth tree are the values, and the weights are the depths at each interval of the depth tree.

38.3.8 timeintervaltools.calculate_depth_density_of_intervals

`timeintervaltools.calculate_depth_density_of_intervals`(*intervals*)

Return a Fraction, of the duration of each interval in the depth tree of *intervals*, multiplied by the depth at that interval, divided by the overall duration of *intervals*.

The depth density of a single interval is 1

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(0, 1)
>>> b = TimeInterval(0, 1)
>>> c = TimeInterval(Fraction(1, 2), 1)
>>> timeintervaltools.calculate_depth_density_of_intervals(a)
Duration(1, 1)
>>> timeintervaltools.calculate_depth_density_of_intervals([a, b])
Duration(2, 1)
>>> timeintervaltools.calculate_depth_density_of_intervals([a, c])
Duration(3, 2)
>>> timeintervaltools.calculate_depth_density_of_intervals([a, b, c])
Duration(5, 2)
```

Return fraction.

38.3.9 timeintervaltools.calculate_depth_density_of_intervals_in_interval

timeintervaltools.**calculate_depth_density_of_intervals_in_interval**(*intervals*, *interval*)

Return a Fraction, of the duration of each interval in the depth tree of *intervals* within *interval*, multiplied by the depth at that interval, divided by the overall duration of *intervals*.

38.3.10 timeintervaltools.calculate_mean_attack_of_intervals

timeintervaltools.**calculate_mean_attack_of_intervals**(*intervals*)

Return Fraction of the average attack offset of *intervals*.

38.3.11 timeintervaltools.calculate_mean_release_of_intervals

timeintervaltools.**calculate_mean_release_of_intervals**(*intervals*)

Return a Fraction of the average release offset of *intervals*.

38.3.12 timeintervaltools.calculate_min_mean_and_max_depth_of_intervals

timeintervaltools.**calculate_min_mean_and_max_depth_of_intervals**(*intervals*)

Return a 3-tuple of the minimum, mean and maximum depth of *intervals*. If *intervals* is empty, return None. "Mean" in this case is a weighted mean, where the durations of the intervals in depth tree of *intervals* are the weights

38.3.13 timeintervaltools.calculate_min_mean_and_max_durations_of_intervals

timeintervaltools.**calculate_min_mean_and_max_durations_of_intervals**(*intervals*)

Return a 3-tuple of the minimum, mean and maximum duration of all intervals in *intervals*. If *intervals* is empty, return None.

38.3.14 timeintervaltools.calculate_sustain_centroid_of_intervals

timeintervaltools.**calculate_sustain_centroid_of_intervals**(*intervals*)

Return a weighted mean, such that the centroid of each interval in *intervals* are the values, and the weights are their durations.

38.3.15 timeintervaltools.clip_interval_durations_to_range

timeintervaltools.**clip_interval_durations_to_range**(*intervals*, *minimum=None*, *maximum=None*)

38.3.16 timeintervaltools.compute_depth_of_intervals

timeintervaltools.**compute_depth_of_intervals**(*intervals*)

Compute a tree whose intervals represent the depth (level of overlap) in each boundary pair of *intervals*:

```
>>> from abjad.tools.timeintervaltools import *
>>> a = TimeInterval(0, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 15)
>>> tree = TimeIntervalTree([a, b, c])
>>> compute_depth_of_intervals(tree)
TimeIntervalTree([
```

```
        TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
        TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
        TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
        TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
        TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1})
    ])
```

Return interval tree.

38.3.17 timeintervaltools.compute_depth_of_intervals_in_interval

timeintervaltools.**compute_depth_of_intervals_in_interval**(*intervals*, *interval*)

Compute a tree whose intervals represent the depth (level of overlap) in each boundary pair of *intervals*, cropped within *interval*:

```
>>> from abjad.tools.timeintervaltools import *
>>> a = TimeInterval(0, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 15)
>>> tree = TimeIntervalTree([a, b, c])
>>> d = TimeInterval(-1, 16)
>>> compute_depth_of_intervals_in_interval(tree, d)
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(0, 1), {'depth': 0}),
    TimeInterval(Offset(0, 1), Offset(3, 1), {'depth': 1}),
    TimeInterval(Offset(3, 1), Offset(6, 1), {'depth': 0}),
    TimeInterval(Offset(6, 1), Offset(9, 1), {'depth': 1}),
    TimeInterval(Offset(9, 1), Offset(12, 1), {'depth': 2}),
    TimeInterval(Offset(12, 1), Offset(15, 1), {'depth': 1}),
    TimeInterval(Offset(15, 1), Offset(16, 1), {'depth': 0})
])
```

Return interval tree.

38.3.18 timeintervaltools.compute_logical_and_of_intervals

timeintervaltools.**compute_logical_and_of_intervals**(*intervals*)

Compute the logical AND of a collection of intervals.

Return TimeIntervalTree.

38.3.19 timeintervaltools.compute_logical_and_of_intervals_in_interval

timeintervaltools.**compute_logical_and_of_intervals_in_interval**(*intervals*, *interval*)

Compute the logical AND of a collection of intervals, cropped within *interval*.

Return TimeIntervalTree.

38.3.20 timeintervaltools.compute_logical_not_of_intervals

timeintervaltools.**compute_logical_not_of_intervals**(*intervals*)

Compute the logical NOT of some collection of intervals.

Return TimeIntervalTree.

38.3.21 timeintervaltools.compute_logical_not_of_intervals_in_interval

timeintervaltools.**compute_logical_not_of_intervals_in_interval**(*intervals*, *interval*)

Compute the logical NOT of some collection of intervals, cropped within *interval*.

Return TimeIntervalTree.

38.3.22 timeintervaltools.compute_logical_or_of_intervals

timeintervaltools.**compute_logical_or_of_intervals**(*intervals*)

Compute the logical OR of a collection of intervals.

Return TimeIntervalTree.

38.3.23 timeintervaltools.compute_logical_or_of_intervals_in_interval

timeintervaltools.**compute_logical_or_of_intervals_in_interval**(*intervals*, *interval*)

Compute the logical OR of a collection of intervals, cropped within *interval*.

Return TimeIntervalTree.

38.3.24 timeintervaltools.compute_logical_xor_of_intervals

timeintervaltools.**compute_logical_xor_of_intervals**(*intervals*)

Compute the logical XOR of a collections of intervals.

Return TimeIntervalTree.

38.3.25 timeintervaltools.compute_logical_xor_of_intervals_in_interval

timeintervaltools.**compute_logical_xor_of_intervals_in_interval**(*intervals*, *interval*)

Compute the logical XOR of a collections of intervals, cropped within *interval*.

Return TimeIntervalTree.

38.3.26 timeintervaltools.concatenate_trees

timeintervaltools.**concatenate_trees**(*trees*, *padding=0*)

Merge all trees in *trees*, offsetting each subsequent tree to start after the previous.

Return TimeIntervalTree.

38.3.27 timeintervaltools.explode_intervals_compactly

timeintervaltools.**explode_intervals_compactly**(*intervals*)

Explode the intervals in *intervals* into n non-overlapping trees, where n is the maximum depth of *intervals*.

The algorithm will attempt to insert the exploded intervals into the lowest-indexed resultant tree with free space.

Return an array of *TimeIntervalTree* instances.

38.3.28 timeintervaltools.explode_intervals_into_n_trees_heuristically

timeintervaltools.**explode_intervals_into_n_trees_heuristically**(*intervals*, *n*)

Explode *intervals* into *n* trees, avoiding overlap when possible, and distributing intervals so as to equalize density across the trees.

Return list of TimeIntervalTree instances.

38.3.29 timeintervaltools.explode_intervals_uncompactly

timeintervaltools.**explode_intervals_uncompactly**(*intervals*)

Explode the intervals in *intervals* into n non-overlapping trees, where n is the maximum depth of *intervals*.

The algorithm will attempt to insert the exploded intervals cyclically, making its insertion attempt at the next resultant tree in the array, rather than always beginning its search from index 0.

Return list of *TimeIntervalTree* instances.

38.3.30 timeintervaltools.fuse_overlapping_intervals

timeintervaltools.**fuse_overlapping_intervals**(*intervals*)

Fuse the overlapping intervals in *intervals* and return an *TimeIntervalTree* of the result

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(0, 10)
>>> b = TimeInterval(5, 15)
>>> c = TimeInterval(15, 25)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.fuse_overlapping_intervals(tree)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(15, 1), {}),
    TimeInterval(Offset(15, 1), Offset(25, 1), {})
])
```

Return TimeIntervalTree.

38.3.31 timeintervaltools.fuse_tangent_or_overlapping_intervals

timeintervaltools.**fuse_tangent_or_overlapping_intervals**(*intervals*)

Fuse all tangent or overlapping intervals and return an *TimeIntervalTree* of the result

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(0, 10)
>>> b = TimeInterval(5, 15)
>>> c = TimeInterval(15, 25)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.fuse_tangent_or_overlapping_intervals(tree)
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(25, 1), {})
])
```

Return TimeIntervalTree.

38.3.32 timeintervaltools.get_all_unique_bounds_in_intervals

timeintervaltools.**get_all_unique_bounds_in_intervals**(*intervals*)

Find all unique starting and ending boundaries in *intervals*.

Return list of Offsets.

38.3.33 timeintervaltools.group_overlapping_intervals_and_yield_groups

timeintervaltools.**group_overlapping_intervals_and_yield_groups**(*intervals*)

Group overlapping intervals in *intervals*.

Yield TimeIntervalTrees.

38.3.34 timeintervaltools.group_tangent_or_overlapping_intervals_and_yield_groups

timeintervaltools.**group_tangent_or_overlapping_intervals_and_yield_groups**(*intervals*)

Group tangent or overlapping intervals in *intervals*.

Yield TimeIntervalTrees.

38.3.35 timeintervaltools.make_monophonic_percussion_score_from_nonoverlapping_inter

timeintervaltools.**make_monophonic_percussion_score_from_nonoverlapping_intervals**(*intervals*, *colorkey=No*

Create a monophonic percussion score from nonoverlapping interval collection *intervals*.

Return Score.

38.3.36 timeintervaltools.make_polyphonic_percussion_score_from_nonoverlapping_trees

timeintervaltools.**make_polyphonic_percussion_score_from_nonoverlapping_trees**(*trees*, *colorkey=None*)

Make a polyphonic percussion score from a collections of non-overlapping trees.

Return LilyPondFile.

38.3.37 timeintervaltools.make_voice_from_nonoverlapping_intervals

timeintervaltools.**make_voice_from_nonoverlapping_intervals**(*intervals*, *colorkey=None*, *pitch=None*)

38.3.38 timeintervaltools.mask_intervals_with_intervals

timeintervaltools.**mask_intervals_with_intervals**(*masked_intervals*, *mask_intervals*)

Clip or remove all intervals in *masked_intervals* outside of the bounds defined in *mask_intervals*, while maintaining *masked_intervals*‘ payload contents

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(0, 10, {'a': 1})
>>> b = TimeInterval(5, 15, {'b': 2})
>>> tree = TimeIntervalTree([a, b])
>>> mask = TimeInterval(4, 11)
>>> timeintervaltools.mask_intervals_with_intervals(tree, mask)
TimeIntervalTree([
    TimeInterval(Offset(4, 1), Offset(10, 1), {'a': 1}),
    TimeInterval(Offset(5, 1), Offset(11, 1), {'b': 2})
])
```

Return TimeIntervalTree.

38.3.39 timeintervaltools.resolve_overlaps_between_nonoverlapping_trees

timeintervaltools.**resolve_overlaps_between_nonoverlapping_trees**(*trees*, *minimum_duration=None*)

Create a nonoverlapping TimeIntervalTree from *trees*. Intervals in higher-indexed trees in *trees* only appear in part or whole where they do not overlap intervals from starter-indexed trees

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeIntervalTree(TimeInterval(0, 4, {'a': 1}))
>>> b = TimeIntervalTree(TimeInterval(1, 5, {'b': 2}))
>>> c = TimeIntervalTree(TimeInterval(2, 6, {'c': 3}))
>>> d = TimeIntervalTree(TimeInterval(1, 3, {'d': 4}))
>>> timeintervaltools.resolve_overlaps_between_nonoverlapping_trees([a, b, c, d])
TimeIntervalTree([
    TimeInterval(Offset(0, 1), Offset(4, 1), {'a': 1}),
    TimeInterval(Offset(4, 1), Offset(5, 1), {'b': 2}),
    TimeInterval(Offset(5, 1), Offset(6, 1), {'c': 3})
])
```

Return TimeIntervalTree.

38.3.40 timeintervaltools.round_interval_bounds_to_nearest_multiple_of_rational

timeintervaltools.**round_interval_bounds_to_nearest_multiple_of_rational**(*intervals*, *duration*)

Round all start and stop offsets of *intervals* to the nearest multiple of *duration*.

If both start and stop of an interval collapse on the same offset, that interval's stop will be adjusted to the next larger multiple of *duration*.

Return TimeIntervalTree.

38.3.41 timeintervaltools.scale_aggregate_duration_by_rational

timeintervaltools.**scale_aggregate_duration_by_rational**(*intervals*, *rational*)

Scale the aggregate duration of all intervals in *intervals* by *rational*, maintaining the original start offset

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.scale_aggregate_duration_by_rational(tree, Fraction(1, 3))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(1, 3), {}),
    TimeInterval(Offset(4, 3), Offset(10, 3), {}),
    TimeInterval(Offset(7, 3), Offset(14, 3), {})
])
```

Return TimeIntervalTree.

38.3.42 timeintervaltools.scale_aggregate_duration_to_rational

timeintervaltools.**scale_aggregate_duration_to_rational**(*intervals*, *rational*)

Scale the aggregate duration of all intervals in *intervals* to *rational*, maintaining the original start offset

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
```

```
>>> timeintervaltools.scale_aggregate_duration_to_rational(tree, Fraction(16, 7))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(-55, 119), {}),
    TimeInterval(Offset(-1, 17), Offset(89, 119), {}),
    TimeInterval(Offset(41, 119), Offset(9, 7), {})
])
```

Return TimeIntervalTree.

38.3.43 timeintervaltools.scale_interval_durations_by_rational

timeintervaltools.**scale_interval_durations_by_rational**(*intervals*, *rational*)
Scale the duration of each interval in *intervals* by *rational*, maintaining their start offsets

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.scale_interval_durations_by_rational(tree, Fraction(6, 5))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(19, 5), {}),
    TimeInterval(Offset(6, 1), Offset(66, 5), {}),
    TimeInterval(Offset(9, 1), Offset(87, 5), {})
])
```

Return TimeIntervalTree.

38.3.44 timeintervaltools.scale_interval_durations_to_rational

timeintervaltools.**scale_interval_durations_to_rational**(*intervals*, *rational*)
Scale the duration of each interval in *intervals* to *rational*, maintaining their start offsets

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.scale_interval_durations_to_rational(tree, Fraction(1, 7))
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(-6, 7), {}),
    TimeInterval(Offset(6, 1), Offset(43, 7), {}),
    TimeInterval(Offset(9, 1), Offset(64, 7), {})
])
```

Return TimeIntervalTree.

38.3.45 timeintervaltools.scale_interval_offsets_by_rational

timeintervaltools.**scale_interval_offsets_by_rational**(*intervals*, *rational*)
Scale the starting offset of each interval in *intervals* by *rational*, maintaining the startest offset in *intervals*

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.scale_interval_offsets_by_rational(tree, Fraction(4, 5))
```

```
TimeIntervalTree([
    TimeInterval(Offset(-1, 1), Offset(3, 1), {}),
    TimeInterval(Offset(23, 5), Offset(53, 5), {}),
    TimeInterval(Offset(7, 1), Offset(14, 1), {})
])
```

Return interval tree.

38.3.46 timeintervaltools.shift_aggregate_offset_by_rational

timeintervaltools.**shift_aggregate_offset_by_rational**(*intervals*, *rational*)
Shift the aggregate offset of *intervals* by *rational*

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.shift_aggregate_offset_by_rational(tree, Fraction(1, 3))
TimeIntervalTree([
    TimeInterval(Offset(-2, 3), Offset(10, 3), {}),
    TimeInterval(Offset(19, 3), Offset(37, 3), {}),
    TimeInterval(Offset(28, 3), Offset(49, 3), {})
])
```

Return TimeIntervalTree.

38.3.47 timeintervaltools.shift_aggregate_offset_to_rational

timeintervaltools.**shift_aggregate_offset_to_rational**(*intervals*, *rational*)
Shift the aggregate offset of *intervals* to *rational*

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.shift_aggregate_offset_to_rational(tree, Fraction(10, 7))
TimeIntervalTree([
    TimeInterval(Offset(10, 7), Offset(38, 7), {}),
    TimeInterval(Offset(59, 7), Offset(101, 7), {}),
    TimeInterval(Offset(80, 7), Offset(129, 7), {})
])
```

Return TimeIntervalTree.

38.3.48 timeintervaltools.split_intervals_at_rationals

timeintervaltools.**split_intervals_at_rationals**(*intervals*, *offsets*)
Split *intervals* at each offset in *offsets*

```
>>> from abjad.tools.timeintervaltools import TimeInterval
>>> from abjad.tools.timeintervaltools import TimeIntervalTree
```

```
>>> a = TimeInterval(-1, 3)
>>> b = TimeInterval(6, 12)
>>> c = TimeInterval(9, 16)
>>> tree = TimeIntervalTree([a, b, c])
>>> timeintervaltools.split_intervals_at_rationals(tree, [1, Fraction(19, 2)])
TimeIntervalTree([
```

```
    TimeInterval(Offset(-1, 1), Offset(1, 1), {}),
    TimeInterval(Offset(1, 1), Offset(3, 1), {}),
    TimeInterval(Offset(6, 1), Offset(19, 2), {}),
    TimeInterval(Offset(9, 1), Offset(19, 2), {}),
    TimeInterval(Offset(19, 2), Offset(12, 1), {}),
    TimeInterval(Offset(19, 2), Offset(16, 1), {})
])
```

Return TimeIntervalTree.

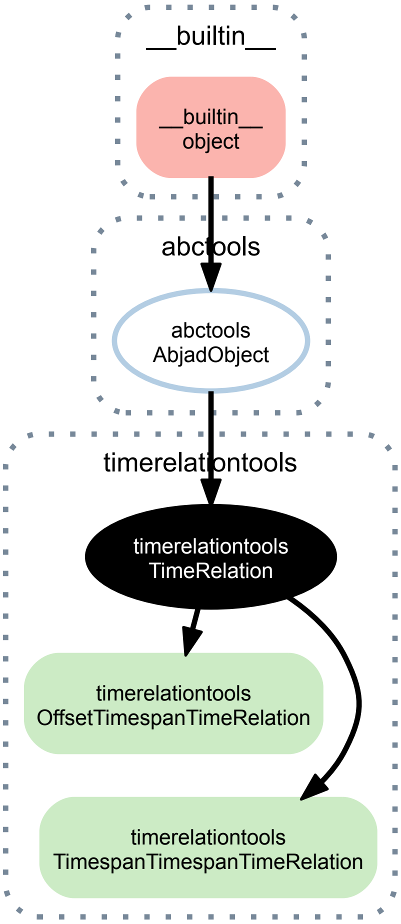
CHAPTER

THIRTYNINE

TIMERELATIONTOOLS

39.1 Abstract Classes

39.1.1 timerelationtools.TimeRelation



class `timerelationtools.`**`TimeRelation`**(*inequality*)

New in version 2.11. Time relation:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1, timespan_2=timespan_2, hold=True)
```

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
```

```
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespan_2=timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(15, 1)
        )
    )
```

Time relations are immutable.

Read-only properties

`TimeRelation.`**inequality**
Time relation inequality.

`TimeRelation.`**is_fully_loaded**
True when both time relation terms are not none. Otherwise false:

```
>>> time_relation.is_fully_loaded
True
```

Return boolean.

`TimeRelation.`**is_fully_unloaded**
True when both time relation terms are none. Otherwise false:

```
>>> time_relation.is_fully_unloaded
False
```

Return boolean.

`TimeRelation.`**storage_format**
Time relation storage format:

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespan_2=timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(15, 1)
        )
    )
```

Return string.

Methods

`TimeRelation.`**new**(***kwargs*)
Inialize new time relation with keyword arguments optionally changed:

```
>>> time_relation = timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = time_relation.new(timespan_1=timespantools.Timespan(0, 5))
```

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

```
>>> z(new_time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        )
    )
```

Return newly constructed time relation.

Special methods

`TimeRelation.__call__()`
Evaluate time relation:

```
>>> time_relation()
True
```

Return boolean.

`TimeRelation.__eq__(expr)`
True when *expr* is a equal-valued time relation. Otherwise false.

Return boolean.

`TimeRelation.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TimeRelation.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TimeRelation.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TimeRelation.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TimeRelation.__ne__(expr)`
Defined equal to the opposite of equality.

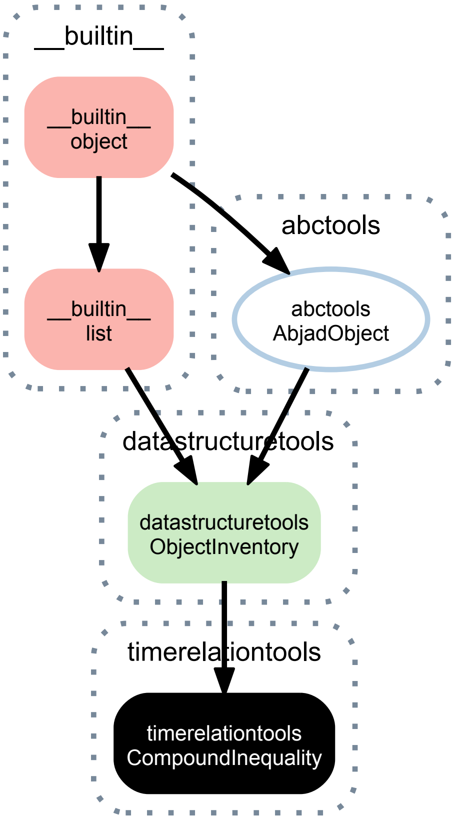
Return boolean.

`TimeRelation.__repr__()`
Interpreter representation of Abjad object.

Return string.

39.2 Concrete Classes

39.2.1 timerelationtools.CompoundInequality



class `timerelationtools.CompoundInequality`(*tokens=None*, *logical_operator='and'*, *name=None*)

New in version 2.12. Compound inequality:

```
>>> compound_inequality = timerelationtools.CompoundInequality([
...    timerelationtools.CompoundInequality([
...        'timespan_1.start_offset <= timespan_2.start_offset',
...        'timespan_2.start_offset < timespan_1.stop_offset'],
...        logical_operator='and'),
...    timerelationtools.CompoundInequality([
...        'timespan_2.start_offset <= timespan_1.start_offset',
...        'timespan_1.start_offset < timespan_2.stop_offset'],
...        logical_operator='and')],
...    logical_operator='or')
```

```
>>> z(compound_inequality)
timerelationtools.CompoundInequality([
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        ),
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
```

```
        ],
        logical_operator='or'
        )
```

Return compound inequality.

Read-only properties

`CompoundInequality.`**`logical_operator`**
Compound inequality logical operator.

`CompoundInequality.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`CompoundInequality.`**`name`**
Read / write name of inventory.

Methods

`CompoundInequality.`**`append`**(*token*)
Change *token* to item and append.

`CompoundInequality.`**`count`**(*value*) → integer – return number of occurrences of value

`CompoundInequality.`**`evaluate`**(*timespan_1_start_offset*, *timespan_1_stop_offset*, *timespan_2_start_offset*, *timespan_2_stop_offset*)

`CompoundInequality.`**`evaluate_offset_inequality`**(*timespan_start*, *timespan_stop*, *offset*)

`CompoundInequality.`**`extend`**(*tokens*)
Change *tokens* to items and extend.

`CompoundInequality.`**`get_offset_indices`**(*timespan_1*, *timespan_2_start_offsets*, *timespan_2_stop_offsets*)

`CompoundInequality.`**`index`**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`CompoundInequality.`**`insert`**()
L.insert(index, object) – insert object before index

`CompoundInequality.`**`pop`**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`CompoundInequality.`**`remove`**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`CompoundInequality.`**`reverse`**()
L.reverse() – reverse *IN PLACE*

`CompoundInequality.`**`sort`**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`CompoundInequality.`**`__add__`**()
x.__add__(y) <==> x+y

CompoundInequality.**__contains__**(*token*)

CompoundInequality.**__delitem__**()
x.__delitem__(y) <==> del x[y]

CompoundInequality.**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

CompoundInequality.**__eq__**()
x.__eq__(y) <==> x==y

CompoundInequality.**__ge__**()
x.__ge__(y) <==> x>=y

CompoundInequality.**__getitem__**()
x.__getitem__(y) <==> x[y]

CompoundInequality.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

CompoundInequality.**__gt__**()
x.__gt__(y) <==> x>y

CompoundInequality.**__iadd__**()
x.__iadd__(y) <==> x+=y

CompoundInequality.**__imul__**()
x.__imul__(y) <==> x*=y

CompoundInequality.**__iter__**() <==> *iter(x)*

CompoundInequality.**__le__**()
x.__le__(y) <==> x<=y

CompoundInequality.**__len__**() <==> *len(x)*

CompoundInequality.**__lt__**()
x.__lt__(y) <==> x<y

CompoundInequality.**__mul__**()
x.__mul__(n) <==> x*n

CompoundInequality.**__ne__**()
x.__ne__(y) <==> x!=y

CompoundInequality.**__repr__**()

CompoundInequality.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

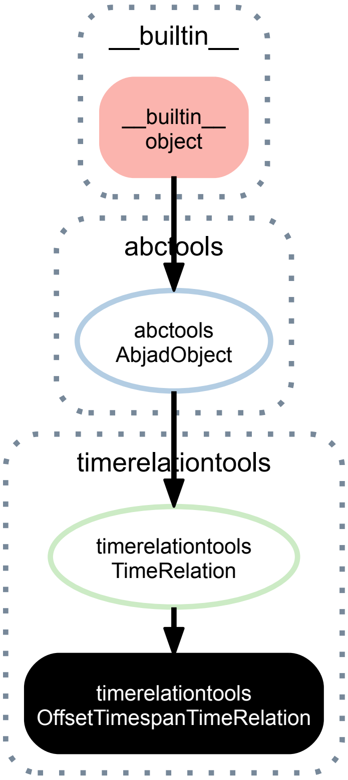
CompoundInequality.**__rmul__**()
x.__rmul__(n) <==> n*x

CompoundInequality.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

CompoundInequality.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

39.2.2 timerelationtools.OffsetTimespanTimeRelation



class `timerelationtools.`**`OffsetTimespanTimeRelation`**(*inequality*, *timespan=None*, *offset=None*)

New in version 2.11. Offset / timespan time relation:

```
>>> offset = Offset(5)
>>> timespan = timespantools.Timespan(0, 10)
>>> time_relation = timerelationtools.offset_happens_during_timespan(
...     offset=offset, timespan=timespan, hold=True)
```

```
>>> z(time_relation)
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan.start <= offset'),
        timerelationtools.SimpleInequality('offset < timespan.stop')
        ],
        logical_operator='and'
        ),
    timespan=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    offset=durationtools.Offset(5, 1)
    )
```

Offset / timespan time relations are immutable.

Read-only properties

`OffsetTimespanTimeRelation.`**`inequality`**
Time relation inequality.

`OffsetTimespanTimeRelation.`**`is_fully_loaded`**
True when *timespan* and *offset* are both not none. Otherwise false:

```
>>> time_relation.is_fully_loaded
True
```

Return boolean.

`OffsetTimespanTimeRelation.`**`is_fully_unloaded`**

True when *timespan* and *offset* are both none. Otherwise false:

```
>>> time_relation.is_fully_unloaded
False
```

Return boolean.

`OffsetTimespanTimeRelation.`**`offset`**

Time relation offset:

```
>>> time_relation.offset
Offset(5, 1)
```

Return offset or none.

`OffsetTimespanTimeRelation.`**`storage_format`**

Time relation storage format:

```
>>> z(time_relation)
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan.start <= offset'),
        timerelationtools.SimpleInequality('offset < timespan.stop')
        ],
        logical_operator='and'
        ),
    timespan=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    offset=durationtools.Offset(5, 1)
    )
```

Return string.

`OffsetTimespanTimeRelation.`**`timespan`**

Time relation timepsan:

```
>>> time_relation.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Return timespan or none.

Methods

`OffsetTimespanTimeRelation.`**`new`**`(**kwargs)`

Inialize new time relation with keyword arguments optionally changed:

```
>>> time_relation = timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = time_relation.new(timespan_1=timespantools.Timespan(0, 5))
```

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

```
>>> z(new_time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        )
    )
```

Return newly constructed time relation.

Special methods

`OffsetTimespanTimeRelation.__call__`(*timespan=None*, *offset=None*)
Evaluate time relation:

```
>>> time_relation()
True
```

Raise value error is either *offset* or *timespan* is none.

Otherwise return boolean.

`OffsetTimespanTimeRelation.__eq__`(*expr*)
True when *expr* equals time relation. Otherwise false:

```
>>> offset = Offset(5)
>>> time_relation_1 = timerelationtools.offset_happens_during_timespan()
>>> time_relation_2 = timerelationtools.offset_happens_during_timespan(
...     offset=offset)
```

```
>>> time_relation_1 == time_relation_1
True
>>> time_relation_1 == time_relation_2
False
>>> time_relation_2 == time_relation_2
True
```

Return boolean.

`OffsetTimespanTimeRelation.__ge__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OffsetTimespanTimeRelation.__gt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`OffsetTimespanTimeRelation.__le__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OffsetTimespanTimeRelation.__lt__`(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

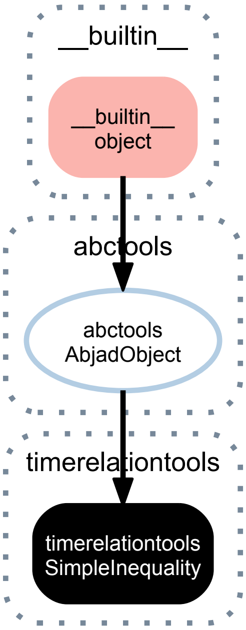
`OffsetTimespanTimeRelation.__ne__`(*expr*)
Defined equal to the opposite of equality.

Return boolean.

OffsetTimespanTimeRelation.**__repr__**()
Interpreter representation of Abjad object.

Return string.

39.2.3 timerelationtools.SimpleInequality



class timerelationtools.**SimpleInequality**(*template*)
New in version 2.12. Simple inequality.

```
>>> template = 'timespan_2.start_offset < timespan_1.start_offset'
>>> simple_inequality = timerelationtools.SimpleInequality(template)
```

```
>>> simple_inequality
SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

Return simple inequality.

Read-only properties

SimpleInequality.**storage_format**
Simple inequality storage format.

```
>>> z(simple_inequality)
timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
```

Return string.

SimpleInequality.**template**
Simple inequality template.

```
>>> simple_inequality.template
'timespan_2.start_offset < timespan_1.start_offset'
```

Return string.

Methods

SimpleInequality.**evaluate**(*timespan_1_start_offset*, *timespan_1_stop_offset*, *timespan_2_start_offset*, *timespan_2_stop_offset*)

`SimpleInequality.`**`evaluate_offset_inequality`**`(`*timespan_start, timespan_stop, offset*`)`

`SimpleInequality.`**`get_offset_indices`**`(`*timespan_1, timespan_2_start_offsets, timespan_2_stop_offsets*`)`

Change simple inequality to offset indices.

Todo

add example.

Return nonnegative integer pair.

Special methods

`SimpleInequality.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`SimpleInequality.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`SimpleInequality.`**`__gt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception

`SimpleInequality.`**`__le__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`SimpleInequality.`**`__lt__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

`SimpleInequality.`**`__ne__`**`(`*expr*`)`

Defined equal to the opposite of equality.

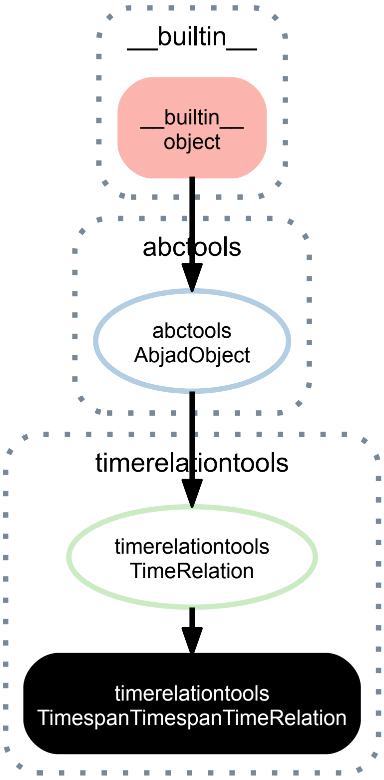
Return boolean.

`SimpleInequality.`**`__repr__`**`()`

Interpreter representation of Abjad object.

Return string.

39.2.4 timerelationtools.TimespanTimespanTimeRelation



class timerelationtools.**TimespanTimespanTimeRelation**(*inequality*, *timespan_1=None*, *timespan_2=None*)

New in version 2.11. Timespan / timespan time relation.

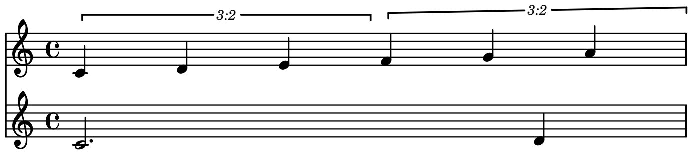
Score for examples:

```
>>> staff_1 = Staff(r"\times 2/3 { c'4 d'4 e'4 } \times 2/3 { f'4 g'4 a'4 }")
>>> staff_2 = Staff("c'2. d'4")
>>> score = Score([staff_1, staff_2])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \times 2/3 {
            c'4
            d'4
            e'4
        }
        \times 2/3 {
            f'4
            g'4
            a'4
        }
    }
    \new Staff {
        c'2.
        d'4
    }
>>
```

```
>>> last_tuplet = staff_1[-1]
>>> long_note = staff_2[0]
```

```
>>> show(score)
```



Example functions calls using the score above:

```
>>> timerelationtools.timespan_2_happens_during_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_intersects_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

```
>>> timerelationtools.timespan_2_is_congruent_to_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_overlaps_all_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_overlaps_start_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

```
>>> timerelationtools.timespan_2_overlaps_stop_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_starts(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_stops(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

Timespan / timespan time relations are immutable.

Read-only properties

`TimespanTimespanTimeRelation.`**`inequality`**

Time relation inequality.

`TimespanTimespanTimeRelation.`**`is_fully_loaded`**

True when *timespan_1* and *timespan_2* are both not none. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1, timespan_2=timespan_2, hold=True)
```

```
>>> time_relation.is_fully_loaded
True
```

Return boolean.

`TimespanTimespanTimeRelation.`**`is_fully_unloaded`**

True when *timespan_1* and *timespan_2* are both none. Otherwise false.

```
>>> time_relation.is_fully_unloaded
False
```

Return boolean.

TimespanTimespanTimeRelation.**storage_format**

Time relation storage format:

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespan_2=timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(15, 1)
        )
    )
```

Return string.

TimespanTimespanTimeRelation.**timespan_1**

Time relation timespan `1`:

```
>>> time_relation.timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Return timespan.

TimespanTimespanTimeRelation.**timespan_2**

Time relation timespan `2`:

```
>>> time_relation.timespan_2
Timespan(start_offset=Offset(5, 1), stop_offset=Offset(15, 1))
```

Return timespan.

Methods

TimespanTimespanTimeRelation.**get_counttime_components**(*counttime_components*)

Get *counttime_components* that satisfy *time_relation*:

```
>>> voice = Voice([Note(i % 36, Duration(1, 4)) for i in range(200)])
>>> timespan_1 = timespantools.Timespan(20, 22)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(timespan_1=timespan_1)
```

```
>>> result = time_relation.get_counttime_components(voice[:])
```

```
>>> for counttime_component in result:
...     counttime_component
Note("af'4")
Note("a'4")
Note("bf'4")
Note("b'4")
Note("c''4")
Note("cs''4")
Note("d''4")
Note("ef''4")
```

```
>>> result.timespan
Timespan(start_offset=Offset(20, 1), stop_offset=Offset(22, 1))
```

counttime_components must belong to a single voice.

counttime_components must be time-contiguous.

Example shown here takes 78355 function calls under r9686.

Return selection.

`TimespanTimespanTimeRelation.`**`get_offset_indices`**`(`*timespan_2_start_offsets, timespan_2_stop_offsets*`)`

Get offset indices that satisfy time relation:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> start_offsets = [note.timespan.start_offset for note in staff]
>>> stop_offsets = [note.timespan.stop_offset for note in staff]
```

Example 1. Notes equal to `staff[0:2]` start during timespan `[0, 3/16)`:

```
>>> timespan_1 = timespantools.Timespan(Offset(0), Offset(3, 16))
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(timespan_1=timespan_1)
>>> time_relation.get_offset_indices(start_offsets, stop_offsets)
(0, 2)
```

Example 2. Notes equal to `staff[2:8]` start after timespan `[0, 3/16)` stops:

```
>>> timespan_1 = timespantools.Timespan(Offset(0), Offset(3, 16))
>>> time_relation = timerelationtools.timespan_2_starts_after_timespan_1_stops(timespan_1=timespan_1)
>>> time_relation.get_offset_indices(start_offsets, stop_offsets)
(2, 8)
```

Return nonnegative integer pair.

`TimespanTimespanTimeRelation.`**`new`**`(`***kwargs*`)`

Inialize new time relation with keyword arguments optionally changed:

```
>>> time_relation = timerelationtools.timespan_2_stops_when_timespan_1_starts()
>>> new_time_relation = time_relation.new(timespan_1=timespantools.Timespan(0, 5))
```

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

```
>>> z(new_time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        )
    )
```

Return newly constructed time relation.

Special methods

`TimespanTimespanTimeRelation.`**`__call__`**`(`*timespan_1=None, timespan_2=None*`)`

Evaluate time relation.

Example 1. Evaluate time relation without substitution:

```
>>> timespan_1 = timespantools.Timespan(5, 15)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1, timespan_2=timespan_2, hold=True)
```

```
>>> z(time_relation)
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        ),
    timespan_1=timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(15, 1)
        ),
    timespan_2=timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    )
```

```
>>> time_relation()
True
```

Example 2. Substitute *timespan_1* during evaluation:

```
>>> new_timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> new_timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

```
>>> time_relation(timespan_1=new_timespan_1)
False
```

Example 3. Substitute *timespan_2* during evaluation:

```
>>> new_timespan_2 = timespantools.Timespan(2, 12)
```

```
>>> new_timespan_2
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(12, 1))
```

```
>>> time_relation(timespan_2=new_timespan_2)
False
```

Example 4. Substitute both *timespan_1* and *timespan_2* during evaluation:

```
>>> time_relation(timespan_1=new_timespan_1, timespan_2=new_timespan_2)
True
```

Raise value error if either *timespan_1* or *timespan_2* is none.

Otherwise return boolean.

`TimespanTimespanTimeRelation.`**`__eq__`**(*expr*)

True when *expr* equals time relation. Otherwise false:

```
>>> timespan = timespantools.Timespan(0, 10)
>>> time_relation_1 = timerelationtools.timespan_2_starts_during_timespan_1()
>>> time_relation_2 = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan)
```

```
>>> time_relation_1 == time_relation_1
True
>>> time_relation_1 == time_relation_2
False
>>> time_relation_2 == time_relation_2
True
```

Return boolean.

`TimespanTimespanTimeRelation.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

TimespanTimespanTimeRelation.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TimespanTimespanTimeRelation.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimespanTimespanTimeRelation.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TimespanTimespanTimeRelation.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

TimespanTimespanTimeRelation.**__repr__**()
Interpreter representation of Abjad object.

Return string.

39.3 Functions

39.3.1 timerelationtools.offset_happens_after_timespan_starts

timerelationtools.**offset_happens_after_timespan_starts**(*timespan=None*, *offset=None*, *hold=False*)
New in version 2.11. Make time relation indicating that *offset* happens after *timespan* starts:

```
>>> z(timerelationtools.offset_happens_after_timespan_starts())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan.start < offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.2 timerelationtools.offset_happens_after_timespan_stops

timerelationtools.**offset_happens_after_timespan_stops**(*timespan=None*, *offset=None*, *hold=False*)
New in version 2.11. Make time relation indicating that *offset* happens after *timespan* stops:

```
>>> z(timerelationtools.offset_happens_after_timespan_stops())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan.stop < offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.3 timerelationtools.offset_happens_before_timespan_starts

timerelationtools.**offset_happens_before_timespan_starts**(*timespan=None*, *offset=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *offset* happens before *timespan* starts:

```
>>> z(timerelationtools.offset_happens_before_timespan_starts())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset < timespan.start')
        ],
        logical_operator='and'
        )
    )
```

Make time relation indicating that offset `1/2` happens before *timespan* starts:

```
>>> offset = durationtools.Offset(1, 2)
```

```
>>> time_relation = timerelationtools.offset_happens_before_timespan_starts(
...     offset=offset)
```

```
>>> z(time_relation)
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset < timespan.start')
        ],
        logical_operator='and'
        ),
    offset=durationtools.Offset(1, 2)
    )
```

Make time relation indicating that *offset* happens before timespan `[2, 8)` starts:

```
>>> timespan = timespantools.Timespan(2, 8)
```

```
>>> time_relation = timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan)
```

```
>>> z(time_relation)
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset < timespan.start')
        ],
        logical_operator='and'
        ),
    timespan=timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(8, 1)
        )
    )
```

Make time relation indicating that offset `1/2` happens before timespan `[2, 8)` starts:

```
>>> time_relation = timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan, offset=offset, hold=True)
```

```
>>> z(time_relation)
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset < timespan.start')
        ],
        logical_operator='and'
        ),
    timespan=timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(8, 1)
        ),
    offset=durationtools.Offset(1, 2)
    )
```

Evaluate time relation indicating that offset `1/2` happens before timespan `[2, 8)` starts:

```
>>> timerelationtools.offset_happens_before_timespan_starts(
...     timespan=timespan, offset=offset, hold=False)
True
```

Return time relation or boolean.

39.3.4 timerelationtools.offset_happens_before_timespan_stops

`timerelationtools.offset_happens_before_timespan_stops`(*timespan=None*, *offset=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *offset* happens before *timespan* stops:

```
>>> z(timerelationtools.offset_happens_before_timespan_stops())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset < timespan.stop')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.5 timerelationtools.offset_happens_during_timespan

`timerelationtools.offset_happens_during_timespan`(*timespan=None*, *offset=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *offset* happens during *timespan*:

```
>>> z(timerelationtools.offset_happens_during_timespan())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan.start <= offset'),
        timerelationtools.SimpleInequality('offset < timespan.stop')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.6 timerelationtools.offset_happens_when_timespan_starts

`timerelationtools.offset_happens_when_timespan_starts`(*timespan=None*, *offset=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *offset* happens when *timespan* starts:

```
>>> z(timerelationtools.offset_happens_when_timespan_starts())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset == timespan.start')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.7 timerelationtools.offset_happens_when_timespan_stops

timerelationtools.**offset_happens_when_timespan_stops**(*timespan=None*, *offset=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *offset* happens when *timespan* stops:

```
>>> z(timerelationtools.offset_happens_when_timespan_stops())
timerelationtools.OffsetTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('offset == timespan.stop')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.8 timerelationtools.timespan_2_contains_timespan_1_improperly

timerelationtools.**timespan_2_contains_timespan_1_improperly**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* contains *timespan_1* improperly:

```
>>> z(timerelationtools.timespan_2_contains_timespan_1_improperly())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.9 timerelationtools.timespan_2_curtails_timespan_1

timerelationtools.**timespan_2_curtails_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* curtails *timespan_1*:

```
>>> z(timerelationtools.timespan_2_curtails_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.stop_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset <= timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.10 timerelationtools.timespan_2_delays_timespan_1

timerelationtools.**timespan_2_delays_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* delays *timespan_1*:

```
>>> z(timerelationtools.timespan_2_delays_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.11 timerelationtools.timespan_2_happens_during_timespan_1

`timerelationtools.`**`timespan_2_happens_during_timespan_1`**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_happens_during_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Evaluate whether timespan `[7/8, 8/8)` happens during timespan `[1/2, 3/2)`:

```
>>> timespan_1 = timespantools.Timespan(Offset(1, 2), Offset(3, 2))
>>> timespan_2 = timespantools.Timespan(Offset(7, 8), Offset(8, 8))
>>> timerelationtools.timespan_2_happens_during_timespan_1(
...     timespan_1=timespan_1, timespan_2=timespan_2)
True
```

Return time relation or boolean.

39.3.12 timerelationtools.timespan_2_intersects_timespan_1

`timerelationtools.`**`timespan_2_intersects_timespan_1`**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* intersects *timespan_1*:

```
>>> z(timerelationtools.timespan_2_intersects_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.CompoundInequality([
            timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
            timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
            ],
            logical_operator='and'
            ),
        timerelationtools.CompoundInequality([
            timerelationtools.SimpleInequality('timespan_2.start_offset <= timespan_1.start_offset'),
            timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
            ],
            logical_operator='and'
            )
        ],
        logical_operator='or'
        )
    )
```

Return time relation or boolean.

39.3.13 timerelationtools.timespan_2_is_congruent_to_timespan_1

timerelationtools.**timespan_2_is_congruent_to_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* is congruent to *timespan_1*:

```
>>> z(timerelationtools.timespan_2_is_congruent_to_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset == timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.14 timerelationtools.timespan_2_overlaps_all_of_timespan_1

timerelationtools.**timespan_2_overlaps_all_of_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* overlaps all of *timespan_1*:

```
>>> z(timerelationtools.timespan_2_overlaps_all_of_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.15 timerelationtools.timespan_2_overlaps_only_start_of_timespan_1

timerelationtools.**timespan_2_overlaps_only_start_of_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_overlaps_only_start_of_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.16 timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1

timerelationtools.**timespan_2_overlaps_only_stop_of_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.17 timerelationtools.timespan_2_overlaps_start_of_timespan_1

timerelationtools.**timespan_2_overlaps_start_of_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* overlaps start of *timespan_1*:

```
>>> z(timerelationtools.timespan_2_overlaps_start_of_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset'),
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.18 timerelationtools.timespan_2_overlaps_stop_of_timespan_1

timerelationtools.**timespan_2_overlaps_stop_of_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* overlaps stop of *timespan_1*:

```
>>> z(timerelationtools.timespan_2_overlaps_stop_of_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset'),
        timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.19 timerelationtools.timespan_2_starts_after_timespan_1_starts

timerelationtools.**timespan_2_starts_after_timespan_1_starts**(*timespan_1=None, timespan_2=None, hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_starts_after_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.20 timerelationtools.timespan_2_starts_after_timespan_1_stops

timerelationtools.**timespan_2_starts_after_timespan_1_stops**(*timespan_1=None, timespan_2=None, hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* starts after *timespan_1* stops:

```
>>> z(timerelationtools.timespan_2_starts_after_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.stop_offset <= timespan_2.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.21 timerelationtools.timespan_2_starts_before_timespan_1_starts

timerelationtools.**timespan_2_starts_before_timespan_1_starts**(*timespan_1=None, timespan_2=None, hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* starts before *timespan_1* starts:

```
>>> z(timerelationtools.timespan_2_starts_before_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.22 timerelationtools.timespan_2_starts_before_timespan_1_stops

timerelationtools.**timespan_2_starts_before_timespan_1_stops**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* starts before *timespan_1* stops:

```
>>> z(timerelationtools.timespan_2_starts_before_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.23 timerelationtools.timespan_2_starts_during_timespan_1

timerelationtools.**timespan_2_starts_during_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* starts during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_starts_during_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset <= timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.start_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Example score:

```
>>> staff_1 = Staff("c'4 d'4 e'4 f'4 g'2 c''2")
>>> staff_2 = Staff("c'2 b'2 a'2 g'2")
>>> score = Score([staff_1, staff_2])
```

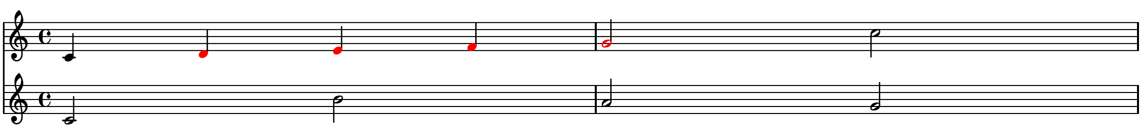
```
>>> start_offsets = [note.timespan.start_offset for note in staff_1]
>>> stop_offsets = [note.timespan.stop_offset for note in staff_1]
```

```
>>> timespan_1 = timespantools.Timespan(Offset(1, 4), Offset(5, 4))
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
>>> start_index, stop_index = time_relation.get_offset_indices(start_offsets, stop_offsets)
```

```
>>> selected_notes = staff_1[start_index:stop_index]
>>> selected_notes
Selection(Note("d'4"), Note("e'4"), Note("f'4"), Note("g'2"))
```

```
>>> labeltools.color_leaves_in_expr(selected_notes, 'red')
```

```
>>> show(score)
```



Return time relation or boolean.

39.3.24 timerelationtools.timespan_2_starts_when_timespan_1_starts

timerelationtools.**timespan_2_starts_when_timespan_1_starts**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* starts when *timespan_1* starts:

```
>>> z(timerelationtools.timespan_2_starts_when_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset == timespan_2.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.25 timerelationtools.timespan_2_starts_when_timespan_1_stops

timerelationtools.**timespan_2_starts_when_timespan_1_stops**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_starts_when_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.start_offset == timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.26 timerelationtools.timespan_2_stops_after_timespan_1_starts

timerelationtools.**timespan_2_stops_after_timespan_1_starts**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* stops after *timespan_1* starts:

```
>>> z(timerelationtools.timespan_2_stops_after_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.27 timerelationtools.timespan_2_stops_after_timespan_1_stops

timerelationtools.**timespan_2_stops_after_timespan_1_stops**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* stops after *timespan_1* stops:

```
>>> z(timerelationtools.timespan_2_stops_after_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.stop_offset < timespan_2.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.28 timerelationtools.timespan_2_stops_before_timespan_1_starts

timerelationtools.**timespan_2_stops_before_timespan_1_starts**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_stops_before_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.29 timerelationtools.timespan_2_stops_before_timespan_1_stops

timerelationtools.**timespan_2_stops_before_timespan_1_stops**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_stops_before_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.30 timerelationtools.timespan_2_stops_during_timespan_1

timerelationtools.**timespan_2_stops_during_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* stops during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_stops_during_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.stop_offset'),
        timerelationtools.SimpleInequality('timespan_2.stop_offset <= timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.31 timerelationtools.timespan_2_stops_when_timespan_1_starts

timerelationtools.**timespan_2_stops_when_timespan_1_starts**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_stops_when_timespan_1_starts())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.start_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.32 timerelationtools.timespan_2_stops_when_timespan_1_stops

timerelationtools.**timespan_2_stops_when_timespan_1_stops**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* happens during *timespan_1*:

```
>>> z(timerelationtools.timespan_2_stops_when_timespan_1_stops())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_2.stop_offset == timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

39.3.33 timerelationtools.timespan_2_trisects_timespan_1

timerelationtools.**timespan_2_trisects_timespan_1**(*timespan_1=None*, *timespan_2=None*, *hold=False*)

New in version 2.11. Make time relation indicating that *timespan_2* trisects *timespan_1*:

```
>>> z(timerelationtools.timespan_2_trisects_timespan_1())
timerelationtools.TimespanTimespanTimeRelation(
    timerelationtools.CompoundInequality([
        timerelationtools.SimpleInequality('timespan_1.start_offset < timespan_2.start_offset'),
        timerelationtools.SimpleInequality('timespan_2.stop_offset < timespan_1.stop_offset')
        ],
        logical_operator='and'
        )
    )
```

Return time relation or boolean.

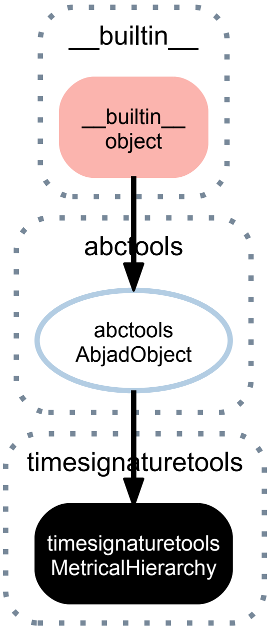
CHAPTER

FORTY

TIMESIGNATURETOOLS

40.1 Concrete Classes

40.1.1 timesignaturetools.MetricalHierarchy



class timesignaturetools.**MetricalHierarchy**(*arg*, *decrease_durations_monotonically=True*)

New in version 2.11. A rhythm tree-based model of nested time signature groupings.

The structure of the tree corresponds to the monotonically increasing sequence of factors of the time signature's numerator.

Each deeper level of the tree divides the previous by the next factor in sequence.

Prime divisions greater than 3 are converted to sequences of 2 and 3 summing to that prime. Hence 5 becomes 3+2 and 7 becomes 3+2+2.

The metrical hierarchy models many parts of the common practice understanding of meter:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
```

```
>>> metrical_hierarchy
MetricalHierarchy('(4/4 (1/4 1/4 1/4 1/4))')
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(4/4 (
    1/4
    1/4
```

```
        1/4
        1/4))
```

```
>>> print timesignaturetools.MetricalHierarchy((3, 4)).pretty_rtm_format
(3/4 (
    1/4
    1/4
    1/4))
```

```
>>> print timesignaturetools.MetricalHierarchy((6, 8)).pretty_rtm_format
(6/8 (
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))))
```

```
>>> print timesignaturetools.MetricalHierarchy((5, 4)).pretty_rtm_format
(5/4 (
    (3/4 (
        1/4
        1/4
        1/4))
    (2/4 (
        1/4
        1/4))))
```

```
>>> print timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=False).pretty_rtm_format
(5/4 (
    (2/4 (
        1/4
        1/4))
    (3/4 (
        1/4
        1/4
        1/4))))
```

```
>>> print timesignaturetools.MetricalHierarchy((12, 8)).pretty_rtm_format
(12/8 (
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))))
```

Return metrical hierarchy object.

Read-only properties

`MetricalHierarchy.`**`decrease_durations_monotonically`**

True if the metrical hierarchy divides large primes into collections of `2` and `3` that decrease monotonically.

Example 1. Metrical hiearchy with durations that increase monotonically:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=False)
```

```
>>> metrical_hierarchy.decrease_durations_monotonically
False
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
    (2/4 (
        1/4
        1/4))
    (3/4 (
        1/4
        1/4
        1/4))))
```

Example 2. Metrical hierarchy with durations that decrease monotonically:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=True)
```

```
>>> metrical_hierarchy.decrease_durations_monotonically
True
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
    (3/4 (
        1/4
        1/4
        1/4))
    (2/4 (
        1/4
        1/4))))
```

Return boolean.

MetricalHierarchy.**denominator**

Beat hierarchy denominator:

```
>>> metrical_hierarchy.denominator
4
```

Return positive integer.

MetricalHierarchy.**depthwise_offset_inventory**

Depthwise inventory of offsets at each grouping level:

```
>>> for depth, offsets in enumerate(metrical_hierarchy.depthwise_offset_inventory):
...     print depth, offsets
0 (Offset(0, 1), Offset(5, 4))
1 (Offset(0, 1), Offset(3, 4), Offset(5, 4))
2 (Offset(0, 1), Offset(1, 4), Offset(1, 2), Offset(3, 4), Offset(1, 1), Offset(5, 4))
```

Return dictionary.

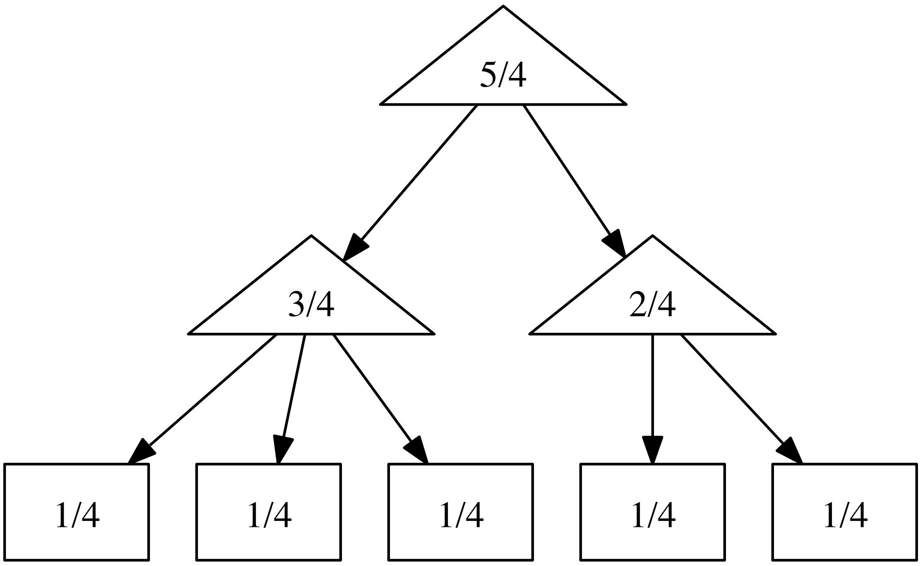
MetricalHierarchy.**graphviz_format**

Graphviz format of hierarchy's root node:

```
>>> print metrical_hierarchy.graphviz_format
digraph G {
    node_0 [label="5/4",
        shape=triangle];
    node_1 [label="3/4",
        shape=triangle];
    node_2 [label="1/4",
        shape=box];
    node_3 [label="1/4",
        shape=box];
    node_4 [label="1/4",
        shape=box];
```

```
        node_5 [label="2/4",
            shape=triangle];
        node_6 [label="1/4",
            shape=box];
        node_7 [label="1/4",
            shape=box];
        node_0 -> node_1;
        node_0 -> node_5;
        node_1 -> node_2;
        node_1 -> node_3;
        node_1 -> node_4;
        node_5 -> node_6;
        node_5 -> node_7;
}
```

```
>>> iotools.graph(metrical_hierarchy)
```



Return string.

`MetricalHierarchy.`**`implied_time_signature`**

Implied time signature:

```
>>> timesignaturetools.MetricalHierarchy((4, 4)).implied_time_signature
TimeSignatureMark((4, 4))
```

Return TimeSignatureMark object.

`MetricalHierarchy.`**`numerator`**

Beat hierarchy numerator:

```
>>> metrical_hierarchy.numerator
5
```

Return positive integer.

`MetricalHierarchy.`**`preprolated_duration`**

Beat hierarchy preprolated_duration:

```
>>> metrical_hierarchy.preprolated_duration
Duration(5, 4)
```

Return preprolated_duration.

`MetricalHierarchy.`**`pretty_rtm_format`**

Beat hieарchy pretty RTM format:

```
>>> print metrical_hierarchy.pretty_rtm_format
(5/4 (
```

```
        (3/4 (
            1/4
            1/4
            1/4))
        (2/4 (
            1/4
            1/4))))
```

Return string.

MetricalHierarchy.**root_node**

Beat hieachy root node:

```
>>> metrical_hierarchy.root_node
RhythmTreeContainer(
    children=(
        RhythmTreeContainer(
            children=(
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 4),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 4),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 4),
                    is_pitched=True
                    )
                ),
            preprolated_duration=NonreducedFraction(3, 4)
            ),
        RhythmTreeContainer(
            children=(
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 4),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 4),
                    is_pitched=True
                    )
                ),
            preprolated_duration=NonreducedFraction(2, 4)
            )
        ),
    preprolated_duration=NonreducedFraction(5, 4)
    )
```

Return rhythm tree node.

MetricalHierarchy.**rtm_format**

Beat hierarchy RTM format:

```
>>> metrical_hierarchy.rtm_format
'(5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4))))'
```

Return string.

MetricalHierarchy.**storage_format**

Beat hierarchy storage format:

```
>>> print metrical_hierarchy.storage_format
timesignaturetools.MetricalHierarchy(
    '(5/4 ((3/4 (1/4 1/4 1/4)) (2/4 (1/4 1/4))))'
    )
```

Return string.

Methods

`MetricalHierarchy.`**`generate_offset_kernel_to_denominator`**(*denominator*, *normalize=True*)

Generate a dictionary of all offsets in a metrical hierarchy up to *denominator*, where the keys are the offsets and the values are the normalized weights of those offsets:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
>>> kernel = metrical_hierarchy.generate_offset_kernel_to_denominator(8)
>>> for offset, weight in sorted(kernel.kernel.iteritems()):
...     print '{}\t{}'.format(offset, weight)
...
0       3/16
1/8     1/16
1/4     1/8
3/8     1/16
1/2     1/8
5/8     1/16
3/4     1/8
7/8     1/16
1       3/16
```

This is useful for testing how strongly a collection of offsets responds to a given metrical hierarchy.

Return dictionary.

Special methods

`MetricalHierarchy.`**`__eq__`**(*expr*)

`MetricalHierarchy.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`MetricalHierarchy.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`MetricalHierarchy.`**`__iter__`**()

Iterate metrical hierarchy:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((5, 4))
```

```
>>> for x in metrical_hierarchy:
...     x
...
(NonreducedFraction(0, 4), NonreducedFraction(1, 4))
(NonreducedFraction(1, 4), NonreducedFraction(2, 4))
(NonreducedFraction(2, 4), NonreducedFraction(3, 4))
(NonreducedFraction(0, 4), NonreducedFraction(3, 4))
(NonreducedFraction(3, 4), NonreducedFraction(4, 4))
(NonreducedFraction(4, 4), NonreducedFraction(5, 4))
(NonreducedFraction(3, 4), NonreducedFraction(5, 4))
(NonreducedFraction(0, 4), NonreducedFraction(5, 4))
```

Yield pairs.

`MetricalHierarchy.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`MetricalHierarchy.`**`__lt__`**(*expr*)

Abjad objects by default do not implement this method.

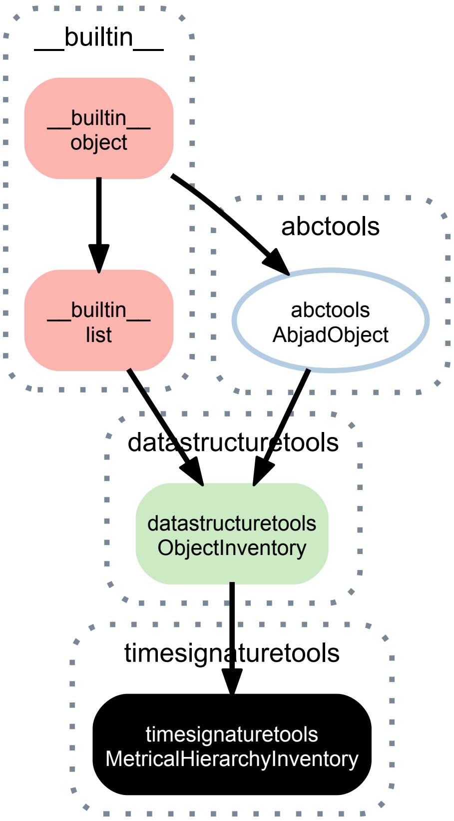
Raise exception.

MetricalHierarchy.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MetricalHierarchy.**__repr__**()

40.1.2 timesignaturetools.MetricalHierarchyInventory



class timesignaturetools.**MetricalHierarchyInventory**(*tokens=None*, *name=None*)
Abjad model of an ordered list of metrical hierarchies:

```
>>> inventory = timesignaturetools.MetricalHierarchyInventory([(4, 4), (3, 4), (6, 8)])
```

```
>>> z(inventory)
timesignaturetools.MetricalHierarchyInventory([
    timesignaturetools.MetricalHierarchy(
        '(4/4 (1/4 1/4 1/4 1/4))'
        ),
    timesignaturetools.MetricalHierarchy(
        '(3/4 (1/4 1/4 1/4))'
        ),
    timesignaturetools.MetricalHierarchy(
        '(6/8 ((3/8 (1/8 1/8 1/8)) (3/8 (1/8 1/8 1/8))))'
        )
    ])
```

MetricalHierarchy inventories implement the list interface and are mutable.

Read-only properties

MetricalHierarchyInventory.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

`MetricalHierarchyInventory.`**name**
Read / write name of inventory.

Methods

`MetricalHierarchyInventory.`**append**(*token*)
Change *token* to item and append.

`MetricalHierarchyInventory.`**count**(*value*) → integer – return number of occurrences of value

`MetricalHierarchyInventory.`**extend**(*tokens*)
Change *tokens* to items and extend.

`MetricalHierarchyInventory.`**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

`MetricalHierarchyInventory.`**insert**()
L.insert(index, object) – insert object before index

`MetricalHierarchyInventory.`**pop**([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`MetricalHierarchyInventory.`**remove**()
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`MetricalHierarchyInventory.`**reverse**()
L.reverse() – reverse *IN PLACE*

`MetricalHierarchyInventory.`**sort**()
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`MetricalHierarchyInventory.`**__add__**()
x.__add__(y) <==> x+y

`MetricalHierarchyInventory.`**__contains__**(*token*)

`MetricalHierarchyInventory.`**__delitem__**()
x.__delitem__(y) <==> del x[y]

`MetricalHierarchyInventory.`**__delslice__**()
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`MetricalHierarchyInventory.`**__eq__**()
x.__eq__(y) <==> x==y

`MetricalHierarchyInventory.`**__ge__**()
x.__ge__(y) <==> x>=y

`MetricalHierarchyInventory.`**__getitem__**()
x.__getitem__(y) <==> x[y]

`MetricalHierarchyInventory.`**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MetricalHierarchyInventory.`**__gt__**()
x.__gt__(y) <==> x>y

MetricalHierarchyInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

MetricalHierarchyInventory.**__imul__**()
x.__imul__(y) <==> x*=y

MetricalHierarchyInventory.**__iter__**() <==> *iter(x)*

MetricalHierarchyInventory.**__le__**()
x.__le__(y) <==> x<=y

MetricalHierarchyInventory.**__len__**() <==> *len(x)*

MetricalHierarchyInventory.**__lt__**()
x.__lt__(y) <==> x<y

MetricalHierarchyInventory.**__mul__**()
x.__mul__(n) <==> x*n

MetricalHierarchyInventory.**__ne__**()
x.__ne__(y) <==> x!=y

MetricalHierarchyInventory.**__repr__**()

MetricalHierarchyInventory.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

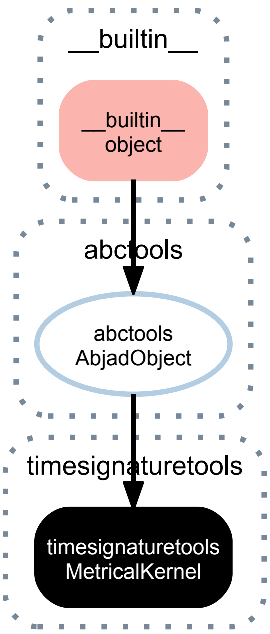
MetricalHierarchyInventory.**__rmul__**()
x.__rmul__(n) <==> n*x

MetricalHierarchyInventory.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

MetricalHierarchyInventory.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

40.1.3 timesignaturetools.MetricalKernel



class timesignaturetools.**MetricalKernel**(*kernel*)
A metrical kernel, or offset-impulse-response-filter:

```
>>> hierarchy = timesignaturetools.MetricalHierarchy((5, 8))
>>> kernel = hierarchy.generate_offset_kernel_to_denominator(8)
>>> kernel
MetricalKernel({
    Offset(0, 1): Multiplier(3, 11),
    Offset(1, 8): Multiplier(1, 11),
    Offset(1, 4): Multiplier(1, 11),
    Offset(3, 8): Multiplier(2, 11),
    Offset(1, 2): Multiplier(1, 11),
    Offset(5, 8): Multiplier(3, 11)
})
```

Call the kernel against an expression from which offsets can be counted to receive an impulse-response:

```
>>> offsets = [(0, 8), (1, 8), (1, 8), (3, 8)]
>>> kernel(offsets)
0.6363636363636364
```

Return *MetricalKernel* instance.

Read-only properties

`MetricalKernel.`**`kernel`**
The kernel datastructure.

Return dict.

`MetricalKernel.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`MetricalKernel.`**`__call__`**(*expr*)

`MetricalKernel.`**`__eq__`**(*expr*)

`MetricalKernel.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MetricalKernel.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`MetricalKernel.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MetricalKernel.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MetricalKernel.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`MetricalKernel.`**`__repr__`**()

40.2 Functions

40.2.1 timesignaturetools.duration_and_possible_denominators_to_time_signature

timesignaturetools.**duration_and_possible_denominators_to_time_signature**(*preprolated_duration*, *denominators=None*, *factor=None*)

Make new time signature equal to *preprolated_duration*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(
...     Duration(3, 2))
TimeSignatureMark((3, 2))
```

Make new time signature equal to *preprolated_duration* with denominator equal to the first possible element in *denominators*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(
...     Duration(3, 2), denominators=[5, 6, 7, 8])
TimeSignatureMark((9, 6))
```

Make new time signature equal to *preprolated_duration* with denominator divisible by *factor*:

```
>>> timesignaturetools.duration_and_possible_denominators_to_time_signature(
...     Duration(3, 2), factor=5)
TimeSignatureMark((15, 10))
```

Note: possibly divide this into two separate functions?

Return new time signature.

40.2.2 timesignaturetools.establish_metrical_hierarchy

timesignaturetools.**establish_metrical_hierarchy**(*components*, *metrical_hierarchy*, *boundary_depth=None*, *maximum_dot_count=None*)

New in version 2.11. Rewrite the contents of tie chains in an expression to match a metrical hierarchy.

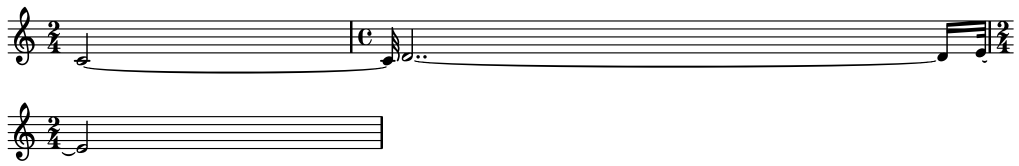
Example 1. Rewrite the contents of a measure in a staff using the default metrical hierarchy for that measure's time signature:

```
>>> parseable = "abj: | 2/4 c'2 ~ || 4/4 c'32 d'2.. ~ d'16 e'32 ~ || 2/4 e'2 |"
```

```
>>> staff = Staff(parseable)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'2.. ~
        d'16
        e'32 ~
    }
    {
        \time 2/4
```

```
            e'2
        }
    }
```

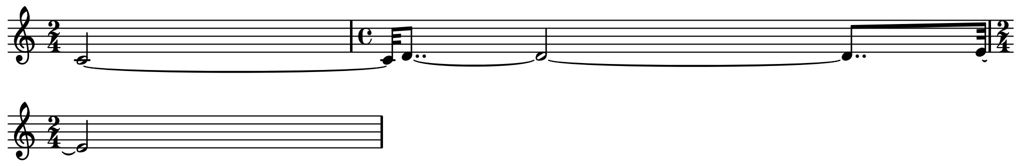
```
>>> show(staff)
```



```
>>> hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
>>> print hierarchy.pretty_rtm_format
(4/4 (
    1/4
    1/4
    1/4
    1/4))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(staff[1][:], hierarchy)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'8.. ~
        d'2 ~
        d'8..
        e'32 ~
    }
    {
        \time 2/4
        e'2
    }
}
```

```
>>> show(staff)
```

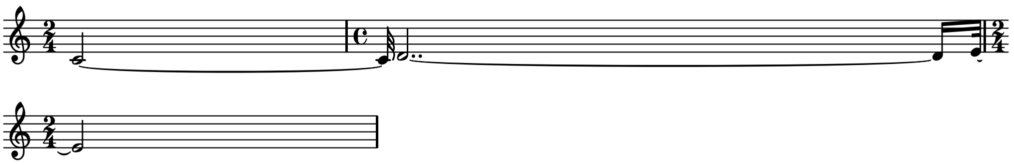


Example 2. Rewrite the contents of a measure in a staff using a custom metrical hierarchy:

```
>>> staff = Staff(parseable)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'2.. ~
        d'16
        e'32 ~
    }
    {
        \time 2/4
        e'2
```

```
        }
    }
```

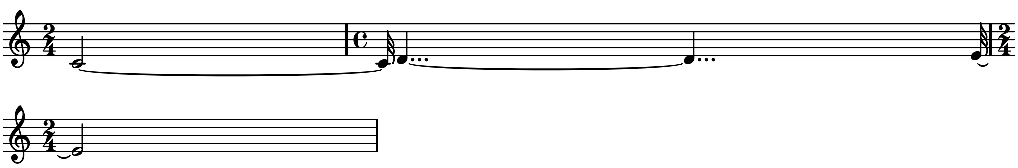
```
>>> show(staff)
```



```
>>> rtm = '(4/4 ((2/4 (1/4 1/4)) (2/4 (1/4 1/4))))'
>>> hierarchy = timesignaturetools.MetricalHierarchy(rtm)
>>> print hierarchy.pretty_rtm_format
(4/4 (
    (2/4 (
        1/4
        1/4))
    (2/4 (
        1/4
        1/4))))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(staff[1][:], hierarchy)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'4... ~
        d'4...
        e'32 ~
    }
    {
        \time 2/4
        e'2
    }
}
```

```
>>> show(staff)
```



Example 3. Limit the maximum number of dots per leaf using *maximum_dot_count*:

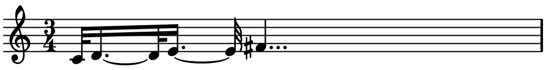
```
>>> parseable = "abj: | 3/4 c'32 d'8 e'8 fs'4... |"
>>> measure = p(parseable)
>>> f(measure)
{
    \time 3/4
    c'32
    d'8
    e'8
    fs'4...
}
```

```
>>> show(measure)
```

Without constraining the *maximum_dot_count*:

```
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'4...
}
```

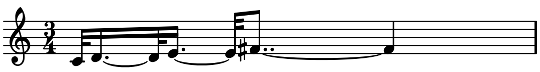
```
>>> show(measure)
```



Constraining the *maximum_dot_count* to *2*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=2)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'8.. ~
    fs'4
}
```

```
>>> show(measure)
```



Constraining the *maximum_dot_count* to *1*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=1)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'16. ~
    fs'8 ~
    fs'4
}
```

```
>>> show(measure)
```

Constraining the *maximum_dot_count* to *0*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=0)
>>> f(measure)
```

```
{
    \time 3/4
    c'32
    d'32 ~
    d'16 ~
    d'32
    e'32 ~
    e'16 ~
    e'32
    fs'32 ~
    fs'16 ~
    fs'8 ~
    fs'4
}
```

```
>>> show(measure)
```

Example 4: Split tie chains at different depths of the *MetricalHierarchy*, if those tie chains cross any offsets at that depth, but do not also both begin and end at any of those offsets.

Consider the default metrical hierarchy for *9/8*:

```
>>> hierarchy = timesignaturetools.MetricalHierarchy((9, 8))
>>> print hierarchy.pretty_rtm_format
(9/8 (
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))))
```

We can establish that hierarchy without specifying a *boundary_depth*:

```
>>> parseable = "abj: | 9/8 c'2 d'2 e'8 |"
>>> measure = p(parseable)
>>> f(measure)
{
    \time 9/8
    c'2
    d'2
    e'8
}
```

```
>>> show(measure)
```

```
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure)
>>> f(measure)
{
    \time 9/8
    c'2
    d'4 ~
    d'4
    e'8
}
```

```
>>> show(measure)
```

With a *boundary_depth* of *1*, tie chains which cross any offsets created by nodes with a depth of *1* in this MetricalHierarchy's rhythm tree - i.e. *0/8*, *3/8*, *6/8* and *9/8* - which do not also begin and end at any of those offsets, will be split:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     boundary_depth=1)
>>> f(measure)
{
    \time 9/8
    c'4. ~
    c'8
    d'4 ~
    d'4
    e'8
}
```

```
>>> show(measure)
```

For this *9/8* hierarchy, and this input notation, A *boundary_depth* of *2* causes no change, as all tie chains already align to multiples of *1/8*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     boundary_depth=2)
>>> f(measure)
{
    \time 9/8
    c'2
    d'4 ~
    d'4
    e'8
}
```

```
>>> show(measure)
```

Example 5. Comparison of *3/4* and *6/8*, at *boundary_depths* of 0 and 1:

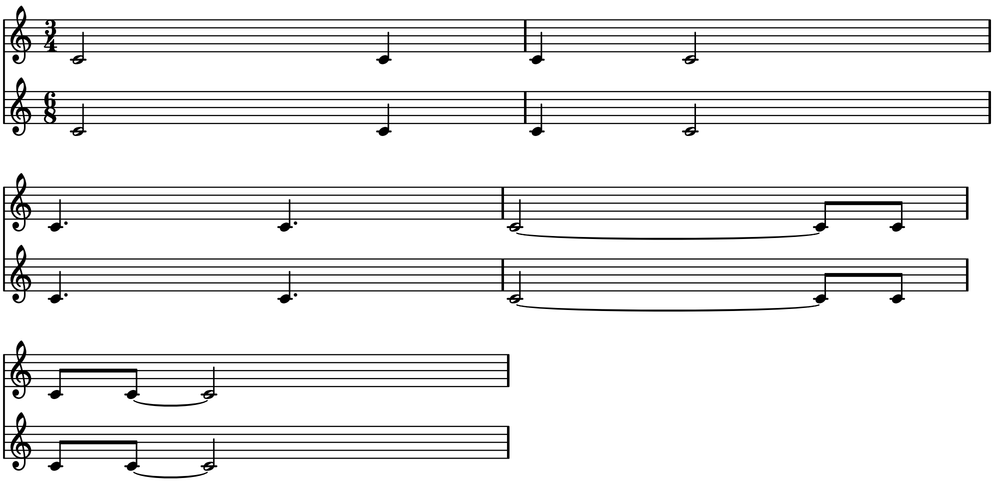
```
>>> triple = "abj: | 3/4 2 4 || 3/4 4 2 || 3/4 4. 4. |"
>>> triple += "| 3/4 2 ~ 8 8 || 3/4 8 8 ~ 2 |"
>>> duples = "abj: | 6/8 2 4 || 6/8 4 2 || 6/8 4. 4. |"
>>> duples += "| 6/8 2 ~ 8 8 || 6/8 8 8 ~ 2 |"
>>> score = Score([Staff(triple), Staff(duples)])
```

In order to see the different time signatures on each staff, we need to move some engravers from the Score context to the Staff context:

```
>>> engravers = ['Timing_translator', 'Time_signature_engraver',
...     'Default_bar_line_engraver']
>>> score.engraver_removals.extend(engravers)
>>> score[0].engraver_consists.extend(engravers)
>>> score[1].engraver_consists.extend(engravers)
>>> f(score)
\new Score \with {
    \remove Timing_translator
    \remove Time_signature_engraver
    \remove Default_bar_line_engraver
```

```
    } <<
        \new Staff \with {
            \consists Timing_translator
            \consists Time_signature_engraver
            \consists Default_bar_line_engraver
        } {
            {
                \time 3/4
                c'2
                c'4
            }
            {
                c'4
                c'2
            }
            {
                c'4.
                c'4.
            }
            {
                c'2 ~
                c'8
                c'8
            }
            {
                c'8
                c'8 ~
                c'2
            }
        }
        \new Staff \with {
            \consists Timing_translator
            \consists Time_signature_engraver
            \consists Default_bar_line_engraver
        } {
            {
                \time 6/8
                c'2
                c'4
            }
            {
                c'4
                c'2
            }
            {
                c'4.
                c'4.
            }
            {
                c'2 ~
                c'8
                c'8
            }
            {
                c'8
                c'8 ~
                c'2
            }
        }
    >>
```

```
>>> show(score)
```

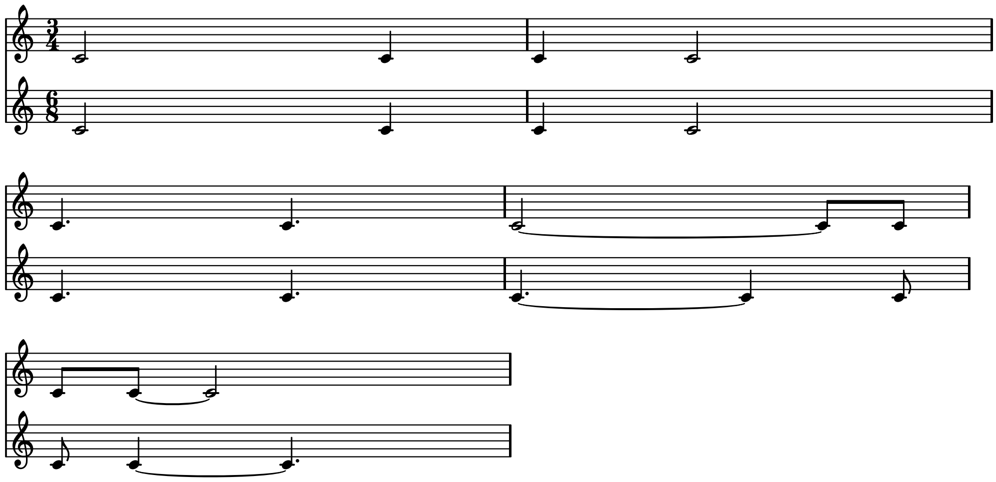


Here we establish a metrical hierarchy without specifying and boundary depth:

```
>>> for measure in iterationtools.iterate_measures_in_expr(score):
...     timesignaturetools.establish_metrical_hierarchy(measure[:], measure)
>>> f(score)
\new Score \with {
    \remove Timing_translator
    \remove Time_signature_engraver
    \remove Default_bar_line_engraver
} <<
    \new Staff \with {
        \consists Timing_translator
        \consists Time_signature_engraver
        \consists Default_bar_line_engraver
    } {
        {
            \time 3/4
            c'2
            c'4
        }
        {
            c'4
            c'2
        }
        {
            c'4.
            c'4.
        }
        {
            c'2 ~
            c'8
            c'8
        }
        {
            c'8
            c'8 ~
            c'2
        }
    }
    \new Staff \with {
        \consists Timing_translator
        \consists Time_signature_engraver
        \consists Default_bar_line_engraver
    } {
        {
            \time 6/8
            c'2
            c'4
        }
        {
            c'4
            c'2
        }
```

```
            {
                c'4.
                c'4.
            }
            {
                c'4. ~
                c'4
                c'8
            }
            {
                c'8
                c'4 ~
                c'4.
            }
        }
>>
```

```
>>> show(score)
```

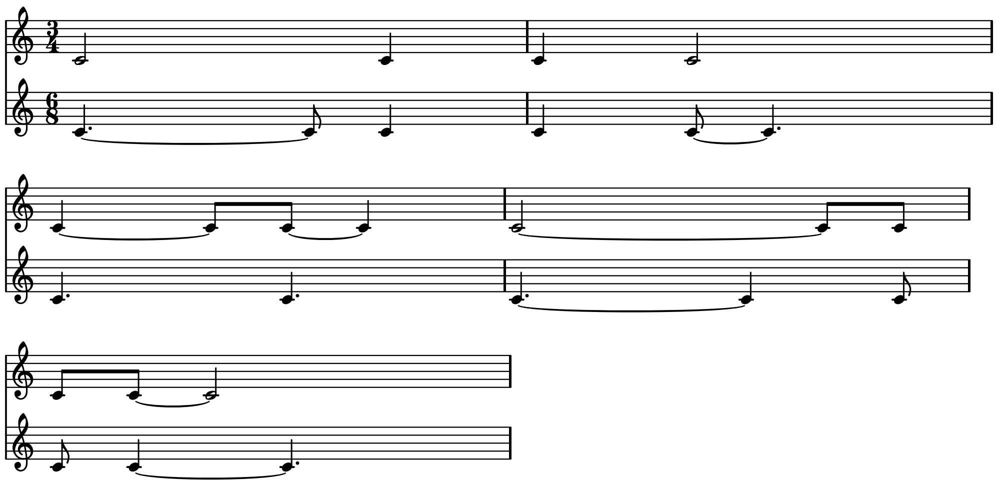


Here we re-establish metrical hierarchy at a boundary depth of *1*:

```
>>> for measure in iterationtools.iterate_measures_in_expr(score):
...     timesignaturetools.establish_metrical_hierarchy(
...         measure[:], measure, boundary_depth=1)
...
>>> f(score)
\new Score \with {
    \remove Timing_translator
    \remove Time_signature_engraver
    \remove Default_bar_line_engraver
} <<
    \new Staff \with {
        \consists Timing_translator
        \consists Time_signature_engraver
        \consists Default_bar_line_engraver
    } {
        {
            \time 3/4
            c'2
            c'4
        }
        {
            c'4
            c'2
        }
        {
            c'4 ~
            c'8
            c'8 ~
            c'4
        }
        {
```

```
                c'2 ~
                c'8
                c'8
            }
            {
                c'8
                c'8 ~
                c'2
            }
        }
        \new Staff \with {
            \consists Timing_translator
            \consists Time_signature_engraver
            \consists Default_bar_line_engraver
        } {
            {
                \time 6/8
                c'4. ~
                c'8
                c'4
            }
            {
                c'4
                c'8 ~
                c'4.
            }
            {
                c'4.
                c'4.
            }
            {
                c'4. ~
                c'4
                c'8
            }
            {
                c'8
                c'4 ~
                c'4.
            }
        }
>>
```

```
>>> show(score)
```



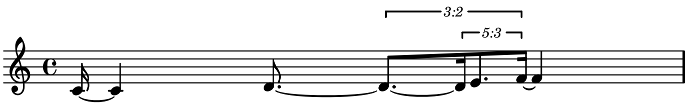
Note that the two time signatures are much more clearly disambiguated above.

Example 6. Establishing metrical hierarchy recursively in measures with nested tuplets:

```
>>> measure = p("abj: | 4/4 c'16 ~ c'4 d'8. ~ " \
...     "2/3 { d'8. ~ 3/5 { d'16 e'8. f'16 ~ } } f'4 |")
>>> f(measure)
```

```
{
    \time 4/4
    c'16 ~
    c'4
    d'8. ~
    \times 2/3 {
        d'8. ~
        \fraction \times 3/5 {
            d'16
            e'8.
            f'16 ~
        }
    }
    f'4
}
```

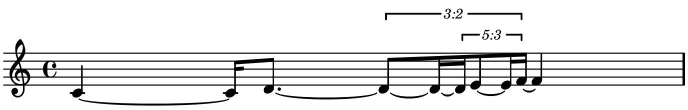
```
>>> show(measure)
```



When establishing a metrical hierarchy on a selection of components which contain containers, like *Tuplets* or *Containers*, *timesignaturetools.establish_metrical_hierarchy()* will recurse into those containers, treating them as measures whose time signature is derived from the preprolated preprolated_duration of the container's contents:

```
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     boundary_depth=1)
>>> f(measure)
{
    \time 4/4
    c'4 ~
    c'16
    d'8. ~
    \times 2/3 {
        d'8 ~
        d'16 ~
        \fraction \times 3/5 {
            d'16
            e'8 ~
            e'16
            f'16 ~
        }
    }
    f'4
}
```

```
>>> show(measure)
```



Operate in place and return none.

40.2.3 timesignaturetools.fit_metrical_hierarchies_to_expr

`timesignaturetools.`**`fit_metrical_hierarchies_to_expr`**(*expr*, *metrical_hierarchies*, *discard_final_orphan_downbeat=True*, *starting_offset=None*, *denominator=32*)

Find the best-matching sequence of metrical hierarchies for the offsets contained in *expr*.

```
>>> metrical_hierarchies = timesignaturetools.MetricalHierarchyInventory(
...     [(3, 4), (4, 4), (5, 4)])
```

Example 1. Matching a series of hypothetical 4/4 measures:

```
>>> expr = [(0, 4), (4, 4), (8, 4), (12, 4), (16, 4)]
>>> for x in timesignaturetools.fit_metrical_hierarchies_to_expr(
...     expr, metrical_hierarchies):
...     print x.implied_time_signature
...
4/4
4/4
4/4
4/4
```

Example 2. Matching a series of hypothetical 5/4 measures:

```
>>> expr = [(0, 4), (3, 4), (5, 4), (10, 4), (15, 4), (20, 4)]
>>> for x in timesignaturetools.fit_metrical_hierarchies_to_expr(
...     expr, metrical_hierarchies):
...     print x.implied_time_signature
...
5/4
5/4
5/4
5/4
```

Offsets are coerced from *expr* via *durationtools.count_offsets_in_expr()*.

MetricalHierarchies are coerced from *metrical_hierarchies* via *MetricalHierarchyInventory*.

Return list.

40.2.4 timesignaturetools.make_gridded_test_rhythm

`timesignaturetools.`**`make_gridded_test_rhythm`**(*grid_length*, *rhythm_number*, *denominator=16*)

New in version 2.11. Make test rhythm number *rhythm_number* that fits *grid_length*.

Return selection of one or more possibly tied notes.

Example 1. The eight test rhythms that fit a length-`4` grid:

```
>>> for rhythm_number in range(8):
...     notes = timesignaturetools.make_gridded_test_rhythm(
...         4, rhythm_number, denominator=4)
...     measure = Measure((4, 4), notes)
...     print '{}\t{}'.format(rhythm_number, measure)
...
0   |4/4 c'1|
1   |4/4 c'2. c'4|
2   |4/4 c'2 c'4 c'4|
3   |4/4 c'2 c'2|
4   |4/4 c'4 c'4 c'2|
5   |4/4 c'4 c'4 c'4 c'4|
6   |4/4 c'4 c'2 c'4|
7   |4/4 c'4 c'2.|
```

Example 2. The sixteenth test rhythms for that a length-`5` grid:

```
>>> for rhythm_number in range(16):
...     notes = timesignaturetools.make_gridded_test_rhythm(
...         5, rhythm_number, denominator=4)
...     measure = Measure((5, 4), notes)
...     print '{}\t{}'.format(rhythm_number, measure)
...
0   |5/4 c'1 ~ c'4|
1   |5/4 c'1 c'4|
2   |5/4 c'2. c'4 c'4|
3   |5/4 c'2. c'2|
```

```
4   |5/4 c'2 c'4 c'2|
5   |5/4 c'2 c'4 c'4 c'4|
6   |5/4 c'2 c'2 c'4|
7   |5/4 c'2 c'2.|
8   |5/4 c'4 c'4 c'2.|
9   |5/4 c'4 c'4 c'2 c'4|
10  |5/4 c'4 c'4 c'4 c'4 c'4|
11  |5/4 c'4 c'4 c'4 c'2|
12  |5/4 c'4 c'2 c'2|
13  |5/4 c'4 c'2 c'4 c'4|
14  |5/4 c'4 c'2. c'4|
15  |5/4 c'4 c'1|
```

Use for testing metrical hierarchy establishment.

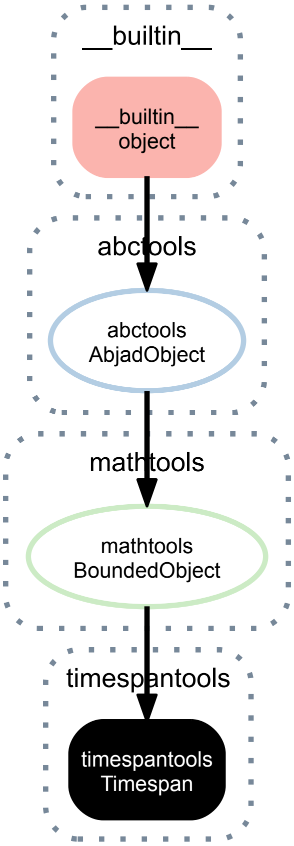
CHAPTER

FORTYONE

TIMESPANTOOLS

41.1 Concrete Classes

41.1.1 timespantools.Timespan



class `timespantools.`**`Timespan`**(*start_offset=NegativeInfinity*, *stop_offset=Infinity*)

Closed-open interval.

Examples:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

Timespans are immutable and treated as value objects.

Read-only properties

`Timespan.axis`
Arithmetic mean of timespan start- and stop-offsets:

```
>>> timespan_1.axis
Offset(5, 1)
```

Return offset.

`Timespan.duration`
Get timespan duration:

```
>>> timespan_1.duration
Duration(10, 1)
```

Return duration.

`Timespan.is_closed`
False for all timespans:

```
>>> timespan_1.is_closed
False
```

Return boolean.

`Timespan.is_half_closed`
True for all timespans:

```
>>> timespan_1.is_half_closed
True
```

Return boolean.

`Timespan.is_half_open`
True for all timespans:

```
>>> timespan_1.is_half_open
True
```

Return boolean.

`Timespan.is_left_closed`
True for all timespans.

```
>>> timespan_1.is_left_closed
True
```

Return boolean.

`Timespan.is_left_open`
False for all timespans.

```
>>> timespan_1.is_left_open
False
```

Return boolean.

`Timespan.is_open`
False for all timespans:

```
>>> timespan_1.is_open
False
```

Return boolean.

`Timespan.is_right_closed`
False for all timespans.

```
>>> timespan_1.is_right_closed
False
```

Return boolean.

Timespan.**is_right_open**
True for all timespans.

```
>>> timespan_1.is_right_open
True
```

Return boolean.

Timespan.**is_well_formed**
True when timespan start offset preceeds timespan stop offset. Otherwise false:

```
>>> timespan_1.is_well_formed
True
```

Return boolean.

Timespan.**offsets**
Timespan offsets:

```
>>> timespan_1.offsets
(Offset(0, 1), Offset(10, 1))
```

Return offset pair.

Timespan.**start_offset**
Timespan start offset:

```
>>> timespan_1.start_offset
Offset(0, 1)
```

Return offset.

Timespan.**stop_offset**
Timespan stop offset:

```
>>> timespan_1.stop_offset
Offset(10, 1)
```

Return offset.

Timespan.**storage_format**
Timespan storage format:

```
>>> z(timespan_1)
timespantools.Timespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(10, 1)
    )
```

Return string.

Methods

Timespan.**contains_timespan_improperly**(*timespan*)
True when timespan contains *timespan* improperly. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.contains_timespan_improperly(timespan_1)
True
>>> timespan_1.contains_timespan_improperly(timespan_2)
True
```

```
>>> timespan_2.contains_timespan_improperly(timespan_1)
False
>>> timespan_2.contains_timespan_improperly(timespan_2)
True
```

Return boolean.

`Timespan.`**`curtails_timespan`**`(`*timespan*`)`

True when timespan curtails *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.curtails_timespan(timespan_1)
False
>>> timespan_1.curtails_timespan(timespan_2)
False
>>> timespan_2.curtails_timespan(timespan_1)
True
>>> timespan_2.curtails_timespan(timespan_2)
False
```

Return boolean.

`Timespan.`**`delays_timespan`**`(`*timespan*`)`

True when timespan delays *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.delays_timespan(timespan_2)
True
>>> timespan_2.delays_timespan(timespan_3)
True
```

Return boolean.

`Timespan.`**`divide_by_ratio`**`(`*ratio*`)`

Divide timespan by *ratio*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> for x in timespan.divide_by_ratio((1, 2, 1)):
...     x
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(3, 4))
Timespan(start_offset=Offset(3, 4), stop_offset=Offset(5, 4))
Timespan(start_offset=Offset(5, 4), stop_offset=Offset(3, 2))
```

Return tuple of newly constructed timespans.

`Timespan.`**`happens_during_timespan`**`(`*timespan*`)`

True when timespan happens during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.happens_during_timespan(timespan_1)
True
>>> timespan_1.happens_during_timespan(timespan_2)
False
>>> timespan_2.happens_during_timespan(timespan_1)
True
>>> timespan_2.happens_during_timespan(timespan_2)
True
```

Return boolean.

`Timespan.`**`intersects_timespan`**`(`*timespan*`)`

True when timespan intersects *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 15)
```

```
>>> timespan_1.intersects_timespan(timespan_1)
True
>>> timespan_1.intersects_timespan(timespan_2)
True
>>> timespan_1.intersects_timespan(timespan_3)
False
```

Return boolean.

`Timespan.`**`is_congruent_to_timespan`**(*timespan*)

True when timespan is congruent to *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.is_congruent_to_timespan(timespan_1)
True
>>> timespan_1.is_congruent_to_timespan(timespan_2)
False
>>> timespan_2.is_congruent_to_timespan(timespan_1)
False
>>> timespan_2.is_congruent_to_timespan(timespan_2)
True
```

Return boolean.

`Timespan.`**`is_tangent_to_timespan`**(*timespan*)

True when timespan is tangent to *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.is_tangent_to_timespan(timespan_1)
False
>>> timespan_1.is_tangent_to_timespan(timespan_2)
True
>>> timespan_2.is_tangent_to_timespan(timespan_1)
True
>>> timespan_2.is_tangent_to_timespan(timespan_2)
False
```

Return boolean.

`Timespan.`**`new`**(***kwargs*)

Create new timespan with *kwargs*:

```
>>> timespan_1.new(stop_offset=Offset(9))
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(9, 1))
```

Return new timespan.

`Timespan.`**`overlaps_all_of_timespan`**(*timespan*)

True when timespan overlaps all of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 6)
>>> timespan_3 = timespantools.Timespan(5, 10)
```

```
>>> timespan_1.overlaps_all_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_all_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_all_of_timespan(timespan_3)
False
```

Return boolean.

Timespan.**overlaps_only_start_of_timespan**(*timespan*)
True when timespan overlaps only start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_only_start_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_2)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_3)
False
>>> timespan_1.overlaps_only_start_of_timespan(timespan_4)
True
```

Return boolean.

Timespan.**overlaps_only_stop_of_timespan**(*timespan*)
True when timespan overlaps only stop of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_3)
False
>>> timespan_1.overlaps_only_stop_of_timespan(timespan_4)
False
```

Return boolean.

Timespan.**overlaps_start_of_timespan**(*timespan*)
True when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_start_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_2)
False
>>> timespan_1.overlaps_start_of_timespan(timespan_3)
True
>>> timespan_1.overlaps_start_of_timespan(timespan_4)
True
```

Return boolean.

Timespan.**overlaps_stop_of_timespan**(*timespan*)
True when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(-5, 5)
>>> timespan_3 = timespantools.Timespan(4, 6)
>>> timespan_4 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.overlaps_stop_of_timespan(timespan_1)
False
>>> timespan_1.overlaps_stop_of_timespan(timespan_2)
True
>>> timespan_1.overlaps_stop_of_timespan(timespan_3)
True
```

```
>>> timespan_1.overlaps_stop_of_timespan(timespan_4)
False
```

Return boolean.

`Timespan.reflect` (*axis=None*)
Reverse timespan about *axis*.

Example 1. Reverse timespan about timespan axis:

```
>>> timespantools.Timespan(3, 6).reflect()
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(6, 1))
```

Example 2. Reverse timespan about arbitrary axis:

```
>>> timespantools.Timespan(3, 6).reflect(axis=Offset(10))
Timespan(start_offset=Offset(14, 1), stop_offset=Offset(17, 1))
```

Emit newly constructed timespan.

`Timespan.scale` (*multiplier*, *anchor=Left*)
Scale timespan by *multiplier*.

```
>>> timespan = timespantools.Timespan(3, 6)
```

Example 1. Scale timespan relative to timespan start offset:

```
>>> timespan.scale(Multiplier(2))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(9, 1))
```

Example 2. Scale timespan relative to timespan stop offset:

```
>>> timespan.scale(Multiplier(2), anchor=Right)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(6, 1))
```

Emit newly constructed timespan.

`Timespan.set_duration` (*duration*)
Set timespan duration to *duration*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.set_duration(Duration(3, 5))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(11, 10))
```

Emit newly constructed timespan.

`Timespan.set_offsets` (*start_offset=None*, *stop_offset=None*)
Set timespan start offset to *start_offset* and stop offset to *stop_offset*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.set_offsets(stop_offset=Offset(7, 8))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(7, 8))
```

Subtract negative *start_offset* from existing stop offset:

```
>>> timespan.set_offsets(start_offset=Offset(-1, 2))
Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 2))
```

Subtract negative *stop_offset* from existing stop offset:

```
>>> timespan.set_offsets(stop_offset=Offset(-1, 2))
Timespan(start_offset=Offset(1, 2), stop_offset=Offset(1, 1))
```

Emit newly constructed timespan.

`Timespan.split_at_offset` (*offset*)
Split into two parts when *offset* happens during timespan:

```
>>> timespan = timespantools.Timespan(0, 5)
```

```
>>> left, right = timespan.split_at_offset(Offset(2))
```

```
>>> left
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(2, 1))
```

```
>>> right
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(5, 1))
```

Otherwise return a copy of timespan:

```
>>> timespan.split_at_offset(Offset(12))
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(5, 1))
```

Return one or two newly constructed timespans.

`Timespan.`**`starts_after_offset`**`(`*offset*`)`
True when timespan overlaps start of *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_after_offset(Offset(-5))
True
>>> timespan_1.starts_after_offset(Offset(0))
False
>>> timespan_1.starts_after_offset(Offset(5))
False
```

Return boolean.

`Timespan.`**`starts_after_timespan_starts`**`(`*timespan*`)`
True when timespan starts after *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_after_timespan_starts(timespan_1)
False
>>> timespan_1.starts_after_timespan_starts(timespan_2)
False
>>> timespan_2.starts_after_timespan_starts(timespan_1)
True
>>> timespan_2.starts_after_timespan_starts(timespan_2)
False
```

Return boolean.

`Timespan.`**`starts_after_timespan_stops`**`(`*timespan*`)`
True when timespan starts after *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 20)
>>> timespan_4 = timespantools.Timespan(15, 25)
```

```
>>> timespan_1.starts_after_timespan_stops(timespan_1)
False
>>> timespan_2.starts_after_timespan_stops(timespan_1)
False
>>> timespan_3.starts_after_timespan_stops(timespan_1)
True
>>> timespan_4.starts_after_timespan_stops(timespan_1)
True
```

Return boolean.

`Timespan.`**`starts_at_offset`**`(`*offset*`)`
True when timespan starts at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_at_offset(Offset(-5))
False
>>> timespan_1.starts_at_offset(Offset(0))
True
>>> timespan_1.starts_at_offset(Offset(5))
False
```

Return boolean.

Timespan.**starts_at_or_after_offset**(*offset*)
True when timespan starts at or after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_at_or_after_offset(Offset(-5))
True
>>> timespan_1.starts_at_or_after_offset(Offset(0))
True
>>> timespan_1.starts_at_or_after_offset(Offset(5))
False
```

Return boolean.

Timespan.**starts_before_offset**(*offset*)
True when timespan starts before *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_before_offset(Offset(-5))
False
>>> timespan_1.starts_before_offset(Offset(0))
False
>>> timespan_1.starts_before_offset(Offset(5))
True
```

Return boolean.

Timespan.**starts_before_or_at_offset**(*offset*)
True when timespan starts before or at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_before_or_at_offset(Offset(-5))
False
>>> timespan_1.starts_before_or_at_offset(Offset(0))
True
>>> timespan_1.starts_before_or_at_offset(Offset(5))
True
```

Return boolean.

Timespan.**starts_before_timespan_starts**(*timespan*)
True when timespan starts before *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_before_timespan_starts(timespan_1)
False
>>> timespan_1.starts_before_timespan_starts(timespan_2)
True
>>> timespan_2.starts_before_timespan_starts(timespan_1)
False
>>> timespan_2.starts_before_timespan_starts(timespan_2)
False
```

Return boolean.

Timespan.**starts_before_timespan_stops**(*timespan*)
True when timespan starts before *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_before_timespan_stops(timespan_1)
True
>>> timespan_1.starts_before_timespan_stops(timespan_2)
True
>>> timespan_2.starts_before_timespan_stops(timespan_1)
True
>>> timespan_2.starts_before_timespan_stops(timespan_2)
True
```

Return boolean.

Timespan.**starts_during_timespan**(*timespan*)
True when timespan starts during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_during_timespan(timespan_1)
True
>>> timespan_1.starts_during_timespan(timespan_2)
False
>>> timespan_2.starts_during_timespan(timespan_1)
True
>>> timespan_2.starts_during_timespan(timespan_2)
True
```

Return boolean.

Timespan.**starts_when_timespan_starts**(*timespan*)
True when timespan starts when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
```

```
>>> timespan_1.starts_when_timespan_starts(timespan_1)
True
>>> timespan_1.starts_when_timespan_starts(timespan_2)
False
>>> timespan_2.starts_when_timespan_starts(timespan_1)
False
>>> timespan_2.starts_when_timespan_starts(timespan_2)
True
```

Return boolean.

Timespan.**starts_when_timespan_stops**(*timespan*)
True when timespan starts when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.starts_when_timespan_stops(timespan_1)
False
>>> timespan_1.starts_when_timespan_stops(timespan_2)
False
>>> timespan_2.starts_when_timespan_stops(timespan_1)
True
>>> timespan_2.starts_when_timespan_stops(timespan_2)
False
```

Return boolean.

Timespan.**stops_after_offset**(*offset*)
True when timespan stops after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.starts_after_offset(Offset(-5))
True
>>> timespan_1.starts_after_offset(Offset(0))
False
>>> timespan_1.starts_after_offset(Offset(5))
False
```

Return boolean.

Timespan.**stops_after_timespan_starts**(*timespan*)
True when timespan stops when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_after_timespan_starts(timespan_1)
True
>>> timespan_1.stops_after_timespan_starts(timespan_2)
False
>>> timespan_2.stops_after_timespan_starts(timespan_1)
True
>>> timespan_2.stops_after_timespan_starts(timespan_2)
True
```

Return boolean.

Timespan.**stops_after_timespan_stops**(*timespan*)
True when timespan stops when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_after_timespan_stops(timespan_1)
False
>>> timespan_1.stops_after_timespan_stops(timespan_2)
False
>>> timespan_2.stops_after_timespan_stops(timespan_1)
True
>>> timespan_2.stops_after_timespan_stops(timespan_2)
False
```

Return boolean.

Timespan.**stops_at_offset**(*offset*)
True when timespan stops at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_at_offset(Offset(-5))
False
>>> timespan_1.stops_at_offset(Offset(0))
False
>>> timespan_1.stops_at_offset(Offset(5))
False
```

Return boolean.

Timespan.**stops_at_or_after_offset**(*offset*)
True when timespan stops at or after *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_at_or_after_offset(Offset(-5))
True
>>> timespan_1.stops_at_or_after_offset(Offset(0))
True
>>> timespan_1.stops_at_or_after_offset(Offset(5))
True
```

Return boolean.

Timespan.**stops_before_offset**(*offset*)
True when timespan stops before *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_before_offset(Offset(-5))
False
>>> timespan_1.stops_before_offset(Offset(0))
False
>>> timespan_1.stops_before_offset(Offset(5))
False
```

Return boolean.

Timespan.**stops_before_or_at_offset**(*offset*)
True when timespan stops before or at *offset*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
```

```
>>> timespan_1.stops_before_or_at_offset(Offset(-5))
False
>>> timespan_1.stops_before_or_at_offset(Offset(0))
False
>>> timespan_1.stops_before_or_at_offset(Offset(5))
False
```

Return boolean.

Timespan.**stops_before_timespan_starts**(*timespan*)
True when timespan stops before *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_before_timespan_starts(timespan_1)
False
>>> timespan_1.stops_before_timespan_starts(timespan_2)
False
>>> timespan_2.stops_before_timespan_starts(timespan_1)
False
>>> timespan_2.stops_before_timespan_starts(timespan_2)
False
```

Return boolean.

Timespan.**stops_before_timespan_stops**(*timespan*)
True when timespan stops before *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_before_timespan_stops(timespan_1)
False
>>> timespan_1.stops_before_timespan_stops(timespan_2)
True
>>> timespan_2.stops_before_timespan_stops(timespan_1)
False
>>> timespan_2.stops_before_timespan_stops(timespan_2)
False
```

Return boolean.

Timespan.**stops_during_timespan**(*timespan*)
True when timespan stops during *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_during_timespan(timespan_1)
True
>>> timespan_1.stops_during_timespan(timespan_2)
False
>>> timespan_2.stops_during_timespan(timespan_1)
False
>>> timespan_2.stops_during_timespan(timespan_2)
True
```

Return boolean.

Timespan.**stops_when_timespan_starts**(*timespan*)

True when timespan stops when *timespan* starts. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_when_timespan_starts(timespan_1)
False
>>> timespan_1.stops_when_timespan_starts(timespan_2)
True
>>> timespan_2.stops_when_timespan_starts(timespan_1)
False
>>> timespan_2.stops_when_timespan_starts(timespan_2)
False
```

Return boolean.

Timespan.**stops_when_timespan_stops**(*timespan*)

True when timespan stops when *timespan* stops. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1.stops_when_timespan_stops(timespan_1)
True
>>> timespan_1.stops_when_timespan_stops(timespan_2)
False
>>> timespan_2.stops_when_timespan_stops(timespan_1)
False
>>> timespan_2.stops_when_timespan_stops(timespan_2)
True
```

Return boolean.

Timespan.**stretch**(*multiplier*, *anchor=None*)

Stretch timespan by *multiplier* relative to *anchor*.

Example 1. Stretch relative to timespan start offset:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2))
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(17, 1))
```

Example 2. Stretch relative to timespan stop offset:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(10))
Timespan(start_offset=Offset(-4, 1), stop_offset=Offset(10, 1))
```

Example 3: Stretch relative to offset prior to timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(0))
Timespan(start_offset=Offset(6, 1), stop_offset=Offset(20, 1))
```

Example 4: Stretch relative to offset after timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(3), Offset(12))
Timespan(start_offset=Offset(-15, 1), stop_offset=Offset(6, 1))
```

Example 5: Stretch relative to offset that happens during timespan:

```
>>> timespantools.Timespan(3, 10).stretch(Multiplier(2), Offset(4))
Timespan(start_offset=Offset(2, 1), stop_offset=Offset(16, 1))
```

Return newly emitted timespan.

`Timespan.`**`translate`**(*translation=None*)
Translate timespan by *translation*.

```
>>> timespan = timespantools.Timespan(5, 10)
```

```
>>> timespan.translate(2)
Timespan(start_offset=Offset(7, 1), stop_offset=Offset(12, 1))
```

Emit newly constructed timespan.

`Timespan.`**`translate_offsets`**(*start_offset_translation=None*, *stop_offset_translation=None*)
Translate timespan start offset by *start_offset_translation* and stop offset by *stop_offset_translation*:

```
>>> timespan = timespantools.Timespan((1, 2), (3, 2))
```

```
>>> timespan.translate_offsets(start_offset_translation=Duration(-1, 8))
Timespan(start_offset=Offset(3, 8), stop_offset=Offset(3, 2))
```

Emit newly constructed timespan.

`Timespan.`**`trisects_timespan`**(*timespan*)
True when timespan trisects *timespan*. Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 6)
```

```
>>> timespan_1.trisects_timespan(timespan_1)
False
>>> timespan_1.trisects_timespan(timespan_2)
False
>>> timespan_2.trisects_timespan(timespan_1)
True
>>> timespan_2.trisects_timespan(timespan_2)
False
```

Return boolean.

Special methods

`Timespan.`**`__and__`**(*expr*)
Logical AND of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1 & timespan_2
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_1 & timespan_3
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_1 & timespan_4
TimespanInventory([])
```

```
>>> timespan_2 & timespan_3
TimespanInventory([])
```

```
>>> timespan_2 & timespan_4
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(12, 1))])
```

```
>>> timespan_3 & timespan_4
TimespanInventory([])
```

Return timespan inventory.

Timespan.**__eq__**(*timespan*)
True when *timespan* is a timespan with equal offsets:

```
>>> timespantools.Timespan(1, 3) == timespantools.Timespan(1, 3)
True
```

Otherwise false:

```
>>> timespantools.Timespan(1, 3) == timespantools.Timespan(2, 3)
False
```

Return boolean.

Timespan.**__ge__**(*expr*)
True when *expr* start offset is greater or equal to timespan start offset:

```
>>> timespan_2 >= timespan_3
True
```

Otherwise false:

```
>>> timespan_1 >= timespan_2
False
```

Return boolean.

Timespan.**__gt__**(*expr*)
True when *expr* start offset is greater than timespan start offset:

```
>>> timespan_2 > timespan_3
True
```

Otherwise false:

```
>>> timespan_1 > timespan_2
False
```

Return boolean.

Timespan.**__le__**(*expr*)
True when *expr* start offset is less than or equal to timespan start offset:

```
>>> timespan_2 <= timespan_3
False
```

Otherwise false:

```
>>> timespan_1 <= timespan_2
True
```

Return boolean.

Timespan.**__len__**()
Defined equal to `1` for all timespans:

```
>>> len(timespan_1)
1
```

Return positive integer.

Timespan.**__lt__**(*expr*)
True when *expr* start offset is less than timespan start offset:

```
>>> timespan_1 < timespan_2
True
```

Otherwise false:

```
>>> timespan_2 < timespan_3
False
```

Return boolean.

`Timespan.__ne__`(*timespan*)

True when *timespan* is not a timespan with equivalent offsets:

```
>>> timespantools.Timespan(1, 3) != timespantools.Timespan(2, 3)
True
```

Otherwise false:

```
>>> timespantools.Timespan(1, 3) != timespantools.Timespan(2/2, (3, 1))
False
```

Return boolean.

`Timespan.__or__`(*expr*)

Logical OR of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> z(timespan_1 | timespan_2)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

```
>>> z(timespan_1 | timespan_3)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

```
>>> z(timespan_1 | timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

```
>>> z(timespan_2 | timespan_3)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

```
>>> z(timespan_2 | timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

```
>>> z(timespan_3 | timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Return timespan inventory.

`Timespan.__repr__()`

Interpreter representation of timespan:

```
>>> timespan_1
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

Return string.

`Timespan.__sub__(`*expr*`)`

Subtract *expr* from timespan:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> timespan_1 - timespan_1
TimespanInventory([])
```

```
>>> timespan_1 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(5, 1))])
```

```
>>> timespan_1 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(2, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_1 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_2 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(12, 1))])
```

```
>>> timespan_2 - timespan_2
TimespanInventory([])
```

```
>>> timespan_2 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(12, 1))])
```

```
>>> timespan_2 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(5, 1), stop_offset=Offset(10, 1))])
```

```
>>> timespan_3 - timespan_3
TimespanInventory([])
```

```
>>> timespan_3 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(0, 1))])
```

```
>>> timespan_3 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_3 - timespan_4
TimespanInventory([Timespan(start_offset=Offset(-2, 1), stop_offset=Offset(2, 1))])
```

```
>>> timespan_4 - timespan_4
TimespanInventory([])
```

```
>>> timespan_4 - timespan_1
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(20, 1))])
```

```
>>> timespan_4 - timespan_2
TimespanInventory([Timespan(start_offset=Offset(12, 1), stop_offset=Offset(20, 1))])
```

```
>>> timespan_4 - timespan_3
TimespanInventory([Timespan(start_offset=Offset(10, 1), stop_offset=Offset(20, 1))])
```

Return timespan inventory.

Timespan.**__xor__**(*expr*)
Logical AND of two timespans:

```
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 12)
>>> timespan_3 = timespantools.Timespan(-2, 2)
>>> timespan_4 = timespantools.Timespan(10, 20)
```

```
>>> z(timespan_1 ^ timespan_2)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

```
>>> z(timespan_1 ^ timespan_3)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(0, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

```
>>> z(timespan_1 ^ timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

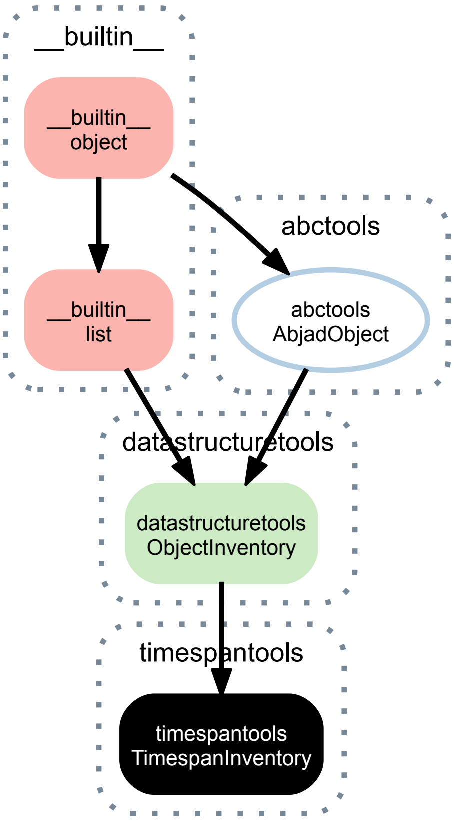
```
>>> z(timespan_2 ^ timespan_3)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

```
>>> z(timespan_2 ^ timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(12, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

```
>>> z(timespan_3 ^ timespan_4)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Return timespan inventory.

41.1.2 timespantools.TimespanInventory



class `timespantools.TimespanInventory`(*tokens=None*, *name=None*)
Timespan inventory.

Example 1:

```
>>> timespan_inventory_1 = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> z(timespan_inventory_1)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 2:

```
>>> timespan_inventory_2 = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(15, 20)])
```

```
>>> z(timespan_inventory_2)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(15, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Example 3. Empty timespan inventory:

```
>>> timespan_inventory_3 = timespantools.TimespanInventory()
```

```
>>> z(timespan_inventory_3)
timespantools.TimespanInventory([])
```

Operations on timespan currently work in place.

Read-only properties

`TimespanInventory.`**`all_are_contiguous`**

True when all timespans are time-contiguous:

```
>>> timespan_inventory_1.all_are_contiguous
True
```

False when timespans not time-contiguous:

```
>>> timespan_inventory_2.all_are_contiguous
False
```

True when empty:

```
>>> timespan_inventory_3.all_are_contiguous
True
```

Return boolean.

`TimespanInventory.`**`all_are_well_formed`**

True when all timespans are well-formed:

```
>>> timespan_inventory_1.all_are_well_formed
True
```

```
>>> timespan_inventory_2.all_are_well_formed
True
```

Also true when empty:

```
>>> timespan_inventory_3.all_are_well_formed
True
```

Otherwise false.

Return boolean.

`TimespanInventory.`**`axis`**

Arithmetic mean of start- and stop-offsets.

```
>>> timespan_inventory_1.axis
Offset(5, 1)
```

```
>>> timespan_inventory_2.axis
Offset(10, 1)
```

None when empty:

```
>>> timespan_inventory_3.axis is None
True
```

Return offset or none.

`TimespanInventory.`**`duration`**

Time from start offset to stop offset:

```
>>> timespan_inventory_1.duration
Duration(10, 1)
```

```
>>> timespan_inventory_2.duration
Duration(20, 1)
```

Zero when empty:

```
>>> timespan_inventory_3.duration
Duration(0, 1)
```

Return duration.

`TimespanInventory.`**`is_sorted`**

True when timespans are in time order:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> timespan_inventory.is_sorted
True
```

Otherwise false:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(6, 10),
...     timespantools.Timespan(3, 6)])
```

```
>>> timespan_inventory.is_sorted
False
```

Return boolean.

`TimespanInventory.`**`start_offset`**

Earliest start offset of any timespan:

```
>>> timespan_inventory_1.start_offset
Offset(0, 1)
```

```
>>> timespan_inventory_2.start_offset
Offset(0, 1)
```

Negative infinity when empty:

```
>>> timespan_inventory_3.start_offset
NegativeInfinity
```

Return offset or none.

`TimespanInventory.`**`stop_offset`**

Latest stop offset of any timespan:

```
>>> timespan_inventory_1.stop_offset
Offset(10, 1)
```

```
>>> timespan_inventory_2.stop_offset
Offset(20, 1)
```

Infinity when empty:

```
>>> timespan_inventory_3.stop_offset
Infinity
```

Return offset or none.

`TimespanInventory.`**`storage_format`**

Storage format of Abjad object.

Return string.

`TimespanInventory.`**`timespan`**

Timespan inventory timespan:

```
>>> timespan_inventory_1.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(10, 1))
```

```
>>> timespan_inventory_2.timespan
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(20, 1))
```

```
>>> timespan_inventory_3.timespan
Timespan(start_offset=NegativeInfinity, stop_offset=Infinity)
```

Return timespan.

Read/write properties

`TimespanInventory.`**`name`**

Read / write name of inventory.

Methods

`TimespanInventory.`**`append`**`(`*token*`)`

Change *token* to item and append.

`TimespanInventory.`**`compute_logical_and`**`()`

Compute logical AND of timespans.

Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10)])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12)])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8)])
```

```
>>> result = timespan_inventory.compute_logical_and()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(8, 1)
        )
    ])
```

Same as setwise intersection.

Operate in place and return timespan inventory.

`TimespanInventory.`**`compute_logical_or`**`()`

Compute logical OR of timespans.

Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory()
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([])
```

Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10)])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12)])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

Example 4:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 2)])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

Example 5:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(-2, 2),
...     timespantools.Timespan(10, 20)])
```

```
>>> result = timespan_inventory.compute_logical_or()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.`**`compute_logical_xor`**`()`

Compute logical XOR of timespans.

Example 1:

```
>>> timespan_inventory = timespantools.TimespanInventory()
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([])
```

Example 2:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 3:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

Example 4:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 2)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(0, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(5, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

Example 5:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(-2, 2),
...     timespantools.Timespan(10, 20)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Example 6:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(4, 8),
...     timespantools.Timespan(2, 6)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(8, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 7:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 10),
...     timespantools.Timespan(0, 10)])
```

```
>>> result = timespan_inventory.compute_logical_xor()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([])
```

Operate in place and return timespan inventory.

`TimespanInventory.`**`count`**(*value*) → integer – return number of occurrences of value

`TimespanInventory.`**`extend`**(*tokens*)
Change *tokens* to items and extend.

`TimespanInventory.`**`get_timespan_that_satisfies_time_relation`**(*time_relation*)
Get timespan that satisifies *time_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 5)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
```

```
>>> timespan_inventory_1.get_timespan_that_satisfies_time_relation(time_relation)
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(6, 1))
```

Return timespan when timespan inventory contains exactly one timespan that satisfies *time_relation*.

Raise exception when timespan inventory contains no timespan that satisfies *time_relation*.

Raise exception when timespan inventory contains more than one timespan that satisfies *time_relation*.

`TimespanInventory.`**`get_timespans_that_satisfy_time_relation`**`(`*time_relation*`)`
Get timespans that satisfy *time_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 8)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
```

```
>>> result = timespan_inventory_1.get_timespans_that_satisfy_time_relation(
...     time_relation)
```

```
>>> z(result)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Return new timespan inventory.

`TimespanInventory.`**`has_timespan_that_satisfies_time_relation`**`(`*time_relation*`)`
True when timespan inventory has timespan that satisfies *time_relation*:

```
>>> timespan_1 = timespantools.Timespan(2, 8)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
```

```
>>> timespan_inventory_1.has_timespan_that_satisfies_time_relation(time_relation)
True
```

Otherwise false:

```
>>> timespan_1 = timespantools.Timespan(10, 20)
>>> time_relation = timerelationtools.timespan_2_starts_during_timespan_1(
...     timespan_1=timespan_1)
```

```
>>> timespan_inventory_1.has_timespan_that_satisfies_time_relation(time_relation)
False
```

Return boolean.

`TimespanInventory.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`TimespanInventory.`**`insert`**`()`
L.insert(index, object) – insert object before index

`TimespanInventory.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`TimespanInventory.`**`reflect`**`(`*axis=None*`)`
Reflect timespans.

Example 1. Reflect timespans about timespan inventory axis:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.reflect()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(4, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(7, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(7, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 2. Reflect timespans about arbitrary axis:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.reflect(axis=Offset(15))
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(20, 1),
        stop_offset=durationtools.Offset(24, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(24, 1),
        stop_offset=durationtools.Offset(27, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(27, 1),
        stop_offset=durationtools.Offset(30, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.`**`remove`**`()`

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`TimespanInventory.`**`remove_degenerate_timespans`**`()`

Remove degenerate timespans:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(5, 5),
...     timespantools.Timespan(5, 10),
...     timespantools.Timespan(5, 25)])
```

```
>>> result = timespan_inventory.remove_degenerate_timespans()
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(25, 1)
        )
    ])
```

Operate in place and return timespan inventory.

TimespanInventory.**repeat_to_stop_offset**(*stop_offset*)

Repeat timespans to *stop_offset*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.repeat_to_stop_offset(15)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(13, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(13, 1),
        stop_offset=durationtools.Offset(15, 1)
        )
    ])
```

Operate in place and return timespan inventory.

TimespanInventory.**reverse**()

L.reverse() – reverse *IN PLACE*

TimespanInventory.**rotate**(*count*)

Rotate by *count* contiguous timespans.

Example 1. Rotate by one timespan to the left:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 4),
...     timespantools.Timespan(4, 10)])
```

```
>>> result = timespan_inventory.rotate(-1)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(7, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(7, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Example 2. Rotate by one timespan to the right:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
```

```
...     timespantools.Timespan(3, 4),
...     timespantools.Timespan(4, 10)])
```

```
>>> result = timespan_inventory.rotate(1)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(9, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(9, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.`**`scale`**(*multiplier*, *anchor=Left*)
Scale timespan by *multiplier* relative to *anchor*.

Example 1. Scale timespans relative to timespan inventory start offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.scale(2)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(9, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(14, 1)
        )
    ])
```

Example 2. Scale timespans relative to timespan inventory stop offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.scale(2, anchor=Right)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-3, 1),
        stop_offset=durationtools.Offset(3, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
```

```
        timespantools.Timespan(
            start_offset=durationtools.Offset(2, 1),
            stop_offset=durationtools.Offset(10, 1)
            )
        ])
```

Operate in place and return timespan inventory.

`TimespanInventory.sort()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

`TimespanInventory.stretch(multiplier, anchor=None)`
Stretch timespans by *multiplier* relative to *anchor*.

Example 1: Stretch timespans relative to timespan inventory start offset:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.stretch(2)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(6, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(6, 1),
        stop_offset=durationtools.Offset(12, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(12, 1),
        stop_offset=durationtools.Offset(20, 1)
        )
    ])
```

Example 2: Stretch timespans relative to arbitrary anchor:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.stretch(2, anchor=Offset(8))
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-8, 1),
        stop_offset=durationtools.Offset(-2, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(4, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(12, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.translate(translation=None)`
Translate timespans by *translation*.

Example 1. Translate timespan by offset `50`:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.translate(50)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(50, 1),
        stop_offset=durationtools.Offset(53, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(53, 1),
        stop_offset=durationtools.Offset(56, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(56, 1),
        stop_offset=durationtools.Offset(60, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.`**`translate_offsets`**(*start_offset_translation=None*, *stop_offset_translation=None*)

Translate timespans by *start_offset_translation* and *stop_offset_translation*.

Example 1. Translate timespan start- and stop-offsets equally:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.translate_offsets(50, 50)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(50, 1),
        stop_offset=durationtools.Offset(53, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(53, 1),
        stop_offset=durationtools.Offset(56, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(56, 1),
        stop_offset=durationtools.Offset(60, 1)
        )
    ])
```

Example 2. Translate timespan stop-offsets only:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10)])
```

```
>>> result = timespan_inventory.translate_offsets(stop_offset_translation=20)
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(23, 1)
        ),
    timespantools.Timespan(
```

```
            start_offset=durationtools.Offset(3, 1),
            stop_offset=durationtools.Offset(26, 1)
            ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(6, 1),
            stop_offset=durationtools.Offset(30, 1)
            )
        ])
```

Operate in place and return timespan inventory.

Special methods

`TimespanInventory.__add__()`
x.__add__(y) <==> x+y

`TimespanInventory.__and__(timespan)`
Keep material that intersects *timespan*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 16),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8)])
```

```
>>> timespan = timespantools.Timespan(5, 10)
>>> result = timespan_inventory & timespan
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(8, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(10, 1)
        )
    ])
```

Operate in place and return timespan inventory.

`TimespanInventory.__contains__(token)`

`TimespanInventory.__delitem__()`
x.__delitem__(y) <==> del x[y]

`TimespanInventory.__delslice__()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`TimespanInventory.__eq__()`
x.__eq__(y) <==> x==y

`TimespanInventory.__ge__()`
x.__ge__(y) <==> x>=y

`TimespanInventory.__getitem__()`
x.__getitem__(y) <==> x[y]

`TimespanInventory.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TimespanInventory.**__gt__**()
x.__gt__(y) <==> x>y

TimespanInventory.**__iadd__**()
x.__iadd__(y) <==> x+=y

TimespanInventory.**__imul__**()
x.__imul__(y) <==> x*=y

TimespanInventory.**__iter__**() <==> *iter(x)*

TimespanInventory.**__le__**()
x.__le__(y) <==> x<=y

TimespanInventory.**__len__**() <==> *len(x)*

TimespanInventory.**__lt__**()
x.__lt__(y) <==> x<y

TimespanInventory.**__mul__**()
x.__mul__(n) <==> x*n

TimespanInventory.**__ne__**()
x.__ne__(y) <==> x!=y

TimespanInventory.**__repr__**()

TimespanInventory.**__reversed__**()
L.__reversed__() – return a reverse iterator over the list

TimespanInventory.**__rmul__**()
x.__rmul__(n) <==> n*x

TimespanInventory.**__setitem__**()
x.__setitem__(i, y) <==> x[i]=y

TimespanInventory.**__setslice__**()
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

TimespanInventory.**__sub__**(*timespan*)
Delete material that intersects *timespan*:

```
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 16),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8)])
```

```
>>> timespan = timespantools.Timespan(5, 10)
>>> result = timespan_inventory - timespan
```

```
>>> z(timespan_inventory)
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(-2, 1),
        stop_offset=durationtools.Offset(5, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(5, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(12, 1)
        ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(16, 1)
        )
    ])
```

Operate in place and return timespan inventory.

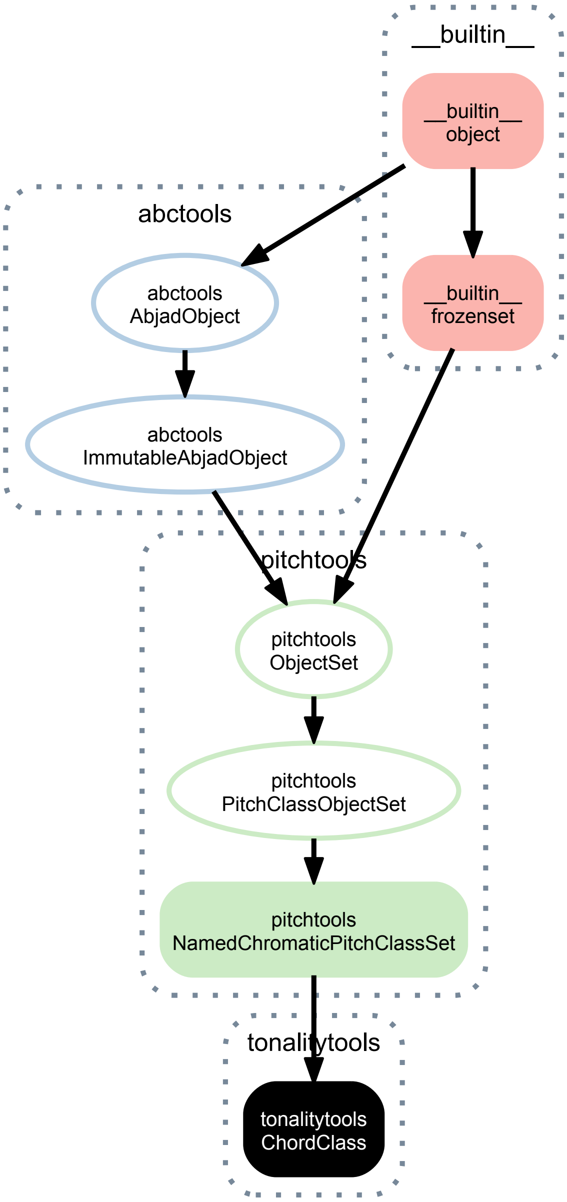
CHAPTER

FORTYTWO

TONALITYTOOLS

42.1 Concrete Classes

42.1.1 tonalitytools.ChordClass



class tonalitytools.**ChordClass**(**args*, ***kwargs*)
New in version 2.0. Abjad model of tonal chords like G 7, G 6/5, G half-diminished 6/5, etc.

Note that notions like G 7 represent an entire *class of* chords because there are many different spacings and registrations of a G 7 chord.

Read-only properties

ChordClass.**bass**

ChordClass.**cardinality**

ChordClass.**extent**

ChordClass.**figured_bass**

ChordClass.**inversion**

ChordClass.**inversion_equivalent_diatonic_interval_class_vector**

ChordClass.**markup**

ChordClass.**named_chromatic_pitch_classes**
Read-only named chromatic pitch-classes:

```
>>> named_chromatic_pitch_class_set = pitchtools.NamedChromaticPitchClassSet(
...     ['gs', 'g', 'as', 'c', 'cs'])
>>> for x in named_chromatic_pitch_class_set.named_chromatic_pitch_classes: x
...
NamedChromaticPitchClass('as')
NamedChromaticPitchClass('c')
NamedChromaticPitchClass('cs')
NamedChromaticPitchClass('g')
NamedChromaticPitchClass('gs')
```

Return tuple.

ChordClass.**numbered_chromatic_pitch_class_set**

ChordClass.**quality_indicator**

ChordClass.**quality_pair**

ChordClass.**root**

ChordClass.**root_string**

ChordClass.**storage_format**
Storage format of Abjad object.

Return string.

Methods

ChordClass.**copy**()
Return a shallow copy of a set.

ChordClass.**difference**()
Return the difference of two or more sets as a new set.

(i.e. all elements that are in this set but not the others.)

ChordClass.**intersection**()
Return the intersection of two or more sets as a new set.

(i.e. elements that are common to all of the sets.)

ChordClass.**isdisjoint**()
Return True if two sets have a null intersection.

ChordClass.**issubset**()
Report whether another set contains this set.

ChordClass.**issuperset**()
Report whether this set contains another set.

ChordClass.**order_by**(*npc_seg*)

ChordClass.**symmetric_difference**()
Return the symmetric difference of two sets as a new set.

(i.e. all elements that are in exactly one of the sets.)

ChordClass.**transpose**()

ChordClass.**union**()
Return the union of sets as a new set.

(i.e. all elements that are in either set.)

Special methods

ChordClass.**__and__**()
x.__and__(y) <==> x&y

ChordClass.**__cmp__**(*y) <==> cmp(x, y*)

ChordClass.**__contains__**()
x.__contains__(y) <==> y in x.

ChordClass.**__eq__**(*arg*)

ChordClass.**__ge__**()
x.__ge__(y) <==> x>=y

ChordClass.**__gt__**()
x.__gt__(y) <==> x>y

ChordClass.**__hash__**()

ChordClass.**__iter__**(*) <==> iter(x*)

ChordClass.**__le__**()
x.__le__(y) <==> x<=y

ChordClass.**__len__**(*) <==> len(x*)

ChordClass.**__lt__**()
x.__lt__(y) <==> x<y

ChordClass.**__ne__**(*arg*)

ChordClass.**__or__**()
x.__or__(y) <==> x|y

ChordClass.**__rand__**()
x.__rand__(y) <==> y&x

ChordClass.**__repr__**()

ChordClass.**__ror__**()
x.__ror__(y) <==> y|x

ChordClass.**__rsub__**()
x.__rsub__(y) <==> y-x

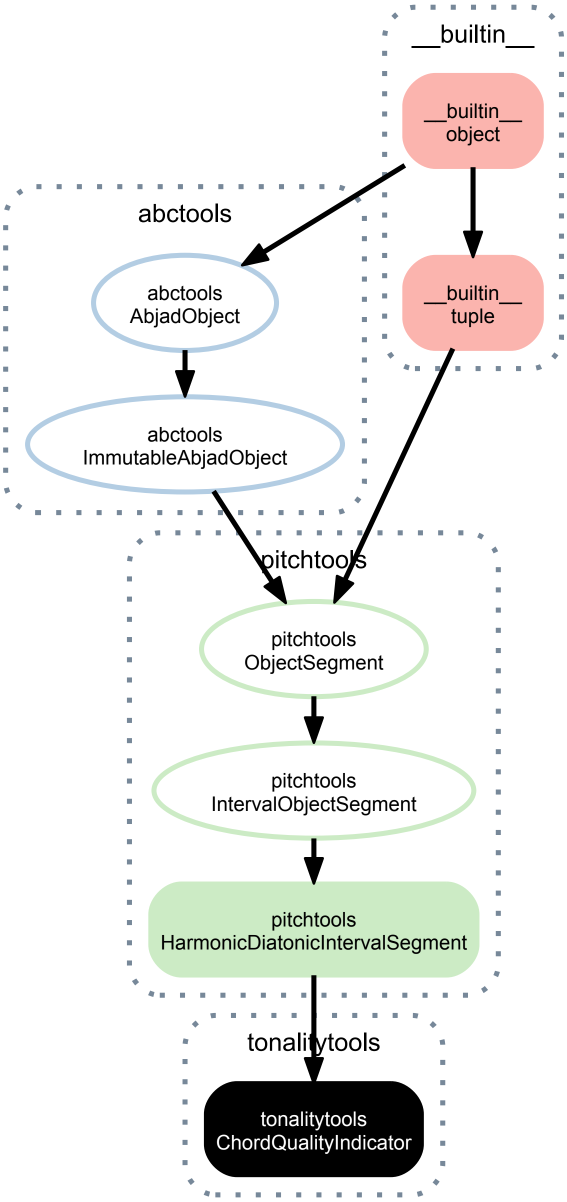
ChordClass.**__rxor__**()
x.__rxor__(y) <==> y^x

ChordClass.**__str__**()

`ChordClass.__sub__()`
x.__sub__(y) <==> x-y

`ChordClass.__xor__()`
x.__xor__(y) <==> x^y

42.1.2 tonalitytools.ChordQualityIndicator



class `tonalitytools.ChordQualityIndicator`(**args*, ***kwargs*)
New in version 2.0. Chord quality indicator.

Read-only properties

`ChordQualityIndicator.cardinality`

`ChordQualityIndicator.extent`

`ChordQualityIndicator.extent_name`

`ChordQualityIndicator.harmonic_chromatic_interval_segment`

ChordQualityIndicator.**interval_classes**

ChordQualityIndicator.**intervals**

ChordQualityIndicator.**inversion**

ChordQualityIndicator.**melodic_chromatic_interval_segment**

ChordQualityIndicator.**melodic_diatonic_interval_segment**

ChordQualityIndicator.**position**

ChordQualityIndicator.**quality_string**

ChordQualityIndicator.**rotation**

ChordQualityIndicator.**storage_format**
Storage format of Abjad object.

Return string.

Methods

ChordQualityIndicator.**count**(*value*) → integer – return number of occurrences of value

ChordQualityIndicator.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

ChordQualityIndicator.**rotate**(*n*)

Special methods

ChordQualityIndicator.**__add__**(*arg*)

ChordQualityIndicator.**__contains__**()
x.__contains__(y) <==> y in x

ChordQualityIndicator.**__copy__**()

ChordQualityIndicator.**__eq__**()
x.__eq__(y) <==> x==y

ChordQualityIndicator.**__ge__**()
x.__ge__(y) <==> x>=y

ChordQualityIndicator.**__getitem__**()
x.__getitem__(y) <==> x[y]

ChordQualityIndicator.**__getslice__**(*start*, *stop*)

ChordQualityIndicator.**__gt__**()
x.__gt__(y) <==> x>y

ChordQualityIndicator.**__hash__**() <==> *hash(x)*

ChordQualityIndicator.**__iter__**() <==> *iter(x)*

ChordQualityIndicator.**__le__**()
x.__le__(y) <==> x<=y

ChordQualityIndicator.**__len__**() <==> *len(x)*

ChordQualityIndicator.**__lt__**()
x.__lt__(y) <==> x<y

ChordQualityIndicator.**__mul__**(*n*)

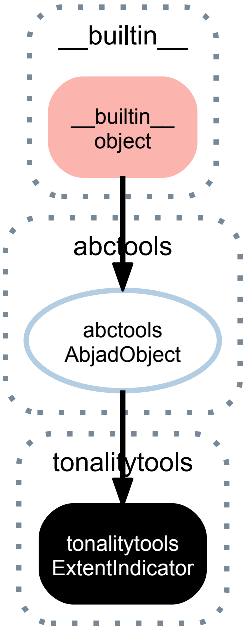
ChordQualityIndicator.**__ne__**()
x.__ne__(y) <==> x!=y

ChordQualityIndicator.**__repr__**()

ChordQualityIndicator.**__rmul__**(*n*)

ChordQualityIndicator.**__str__**()

42.1.3 tonalitytools.ExtentIndicator



class tonalitytools.**ExtentIndicator**(*arg*)

New in version 2.0. Indicator of chord extent, such as triad, seventh chord, ninth chord, etc.

Value object that can not be changed after instantiation.

Read-only properties

ExtentIndicator.**name**

ExtentIndicator.**number**

ExtentIndicator.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

ExtentIndicator.**__eq__**(*arg*)

ExtentIndicator.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ExtentIndicator.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

ExtentIndicator.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

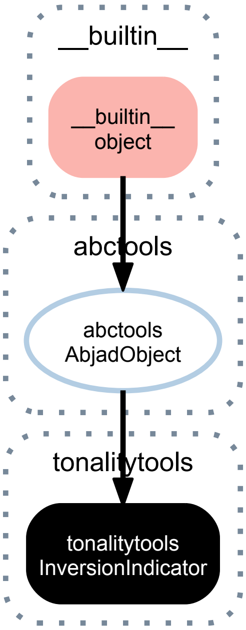
ExtentIndicator.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ExtentIndicator.**__ne__**(*arg*)

ExtentIndicator.**__repr__**()

42.1.4 tonalitytools.InversionIndicator



class tonalitytools.**InversionIndicator**(*arg=0*)
New in version 2.0. Indicator of the inversion of tertian chords: 5, 63, 64 and also 7, 65, 43, 42, etc. Also root position, first, second, third inversions, etc.

Value object that can not be changed once initialized.

Read-only properties

InversionIndicator.**name**

InversionIndicator.**number**

InversionIndicator.**storage_format**
Storage format of Abjad object.

Return string.

InversionIndicator.**title**

Methods

InversionIndicator.**extent_to_figured_bass_string**(*extent*)

Special methods

InversionIndicator.**__eq__**(*arg*)

InversionIndicator.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InversionIndicator.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

InversionIndicator.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

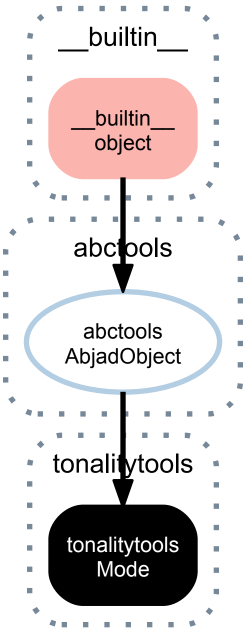
InversionIndicator.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InversionIndicator.**__ne__**(*arg*)

InversionIndicator.**__repr__**()

42.1.5 tonalitytools.Mode



class tonalitytools.**Mode**(*arg*)
New in version 2.0. Diatonic mode. Can be extended for nondiatonic mode.

Modes with different ascending and descending forms not yet implemented.

Read-only properties

Mode.**melodic_diatonic_interval_segment**

Mode.**mode_name**

Mode.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

Mode.**__eq__**(*arg*)

Mode.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Mode.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Mode.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Mode.**__len__**()

Mode.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

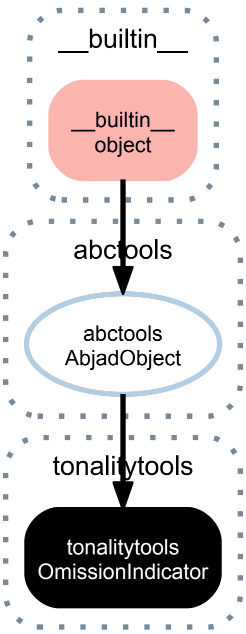
Raise exception.

Mode.**__ne__**(*arg*)

Mode.**__repr__**()

Mode.**__str__**()

42.1.6 tonalitytools.OmissionIndicator



class tonalitytools.**OmissionIndicator**
New in version 2.0. Indicator of missing chord tones.

Value object that can not be chnaged after instantiation.

Read-only properties

`OmissionIndicator.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`OmissionIndicator.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`OmissionIndicator.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`OmissionIndicator.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`OmissionIndicator.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`OmissionIndicator.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

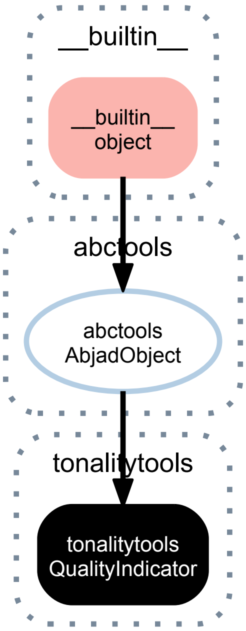
`OmissionIndicator.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`OmissionIndicator.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

42.1.7 tonalitytools.QualityIndicator



class `tonalitytools.`**`QualityIndicator`**(*quality_string*)

New in version 2.0. Indicator of chord quality, such as major, minor, dominant, diminished, etc.

Value object that can not be changed after instantiation.

Read-only properties

`QualityIndicator.`**`is_uppercase`**

`QualityIndicator.`**`quality_string`**

`QualityIndicator.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`QualityIndicator.`**`__eq__`**(*arg*)

`QualityIndicator.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`QualityIndicator.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`QualityIndicator.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`QualityIndicator.`**`__lt__`**(*expr*)

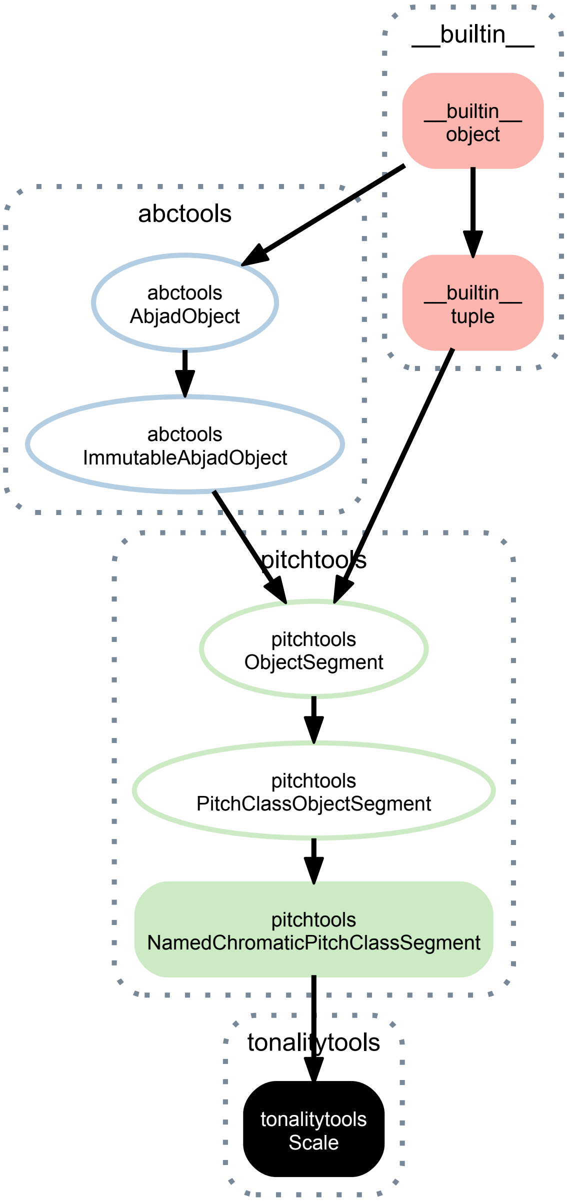
Abjad objects by default do not implement this method.

Raise exception.

`QualityIndicator.`**`__ne__`**(*arg*)

QualityIndicator.**__repr__**()

42.1.8 tonalitytools.Scale



class tonalitytools.**Scale**(**args*, ***kwargs*)

New in version 2.0. Abjad model of diatonic scale.

Read-only properties

Scale.**diatonic_interval_class_segment**

Scale.**dominant**

Scale.**inversion_equivalent_diatonic_interval_class_segment**

Scale.**key_signature**

Scale.**leading_tone**

Scale.**mediant**

Scale.**named_chromatic_pitch_class_set**

Scale.**named_chromatic_pitch_classes**

Scale.**numbered_chromatic_pitch_class_segment**

Scale.**numbered_chromatic_pitch_class_set**

Scale.**numbered_chromatic_pitch_classes**

Scale.**storage_format**
Storage format of Abjad object.

Return string.

Scale.**subdominant**

Scale.**submediant**

Scale.**superdominant**

Scale.**tonic**

Methods

Scale.**count**(*value*) → integer – return number of occurrences of value

Scale.**create_named_chromatic_pitch_set_in_pitch_range**(*pitch_range*)

Scale.**index**(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Scale.**is_equivalent_under_transposition**(*arg*)

Scale.**named_chromatic_pitch_class_to_scale_degree**(**args*)

Scale.**retrograde**()

Scale.**rotate**(*n*)

Scale.**scale_degree_to_named_chromatic_pitch_class**(**args*)

Scale.**transpose**(*melodic_diatonic_interval*)

Special methods

Scale.**__add__**(*arg*)

Scale.**__contains__**()
x.__contains__(y) <==> y in x

Scale.**__eq__**()
x.__eq__(y) <==> x==y

Scale.**__ge__**()
x.__ge__(y) <==> x>=y

Scale.**__getitem__**()
x.__getitem__(y) <==> x[y]

Scale.**__getslice__**(*start*, *stop*)

Scale.**__gt__**()
x.__gt__(y) <==> x>y

Scale.**__hash__**() <==> *hash(x)*

Scale.**__iter__**() <==> *iter(x)*

Scale.**__le__**()
    x.__le__(y) <==> x<=y

Scale.**__len__**() <==> *len(x)*

Scale.**__lt__**()
    x.__lt__(y) <==> x<y

Scale.**__mul__**(*n*)

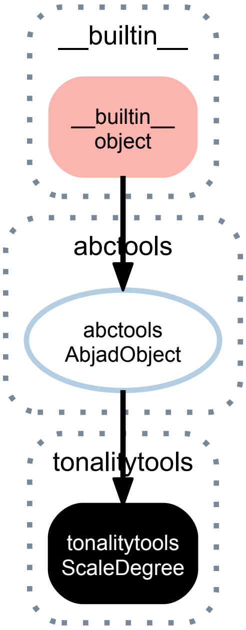
Scale.**__ne__**()
    x.__ne__(y) <==> x!=y

Scale.**__repr__**()

Scale.**__rmul__**(*n*)

Scale.**__str__**()

42.1.9 tonalitytools.ScaleDegree



class tonalitytools.**ScaleDegree**(**args*)
    New in version 2.0. Abjad model of diatonic scale degrees 1, 2, 3, 4, 5, 6, 7 and also chromatic alterations including flat-2, flat-3, flat-6, etc.

Read-only properties

ScaleDegree.**accidental**
    Read-only accidental applied to scale degree.

ScaleDegree.**name**
    Read-only name of scale degree.

ScaleDegree.**number**
    Read-only number of diatonic scale degree from 1 to 7, inclusive.

ScaleDegree.**roman_numeral_string**

ScaleDegree.**storage_format**
    Storage format of Abjad object.

    Return string.

`ScaleDegree.`**`symbolic_string`**

`ScaleDegree.`**`title_string`**

Methods

`ScaleDegree.`**`apply_accidental`**(*accidental*)
Apply accidental to self and emit new instance.

Special methods

`ScaleDegree.`**`__eq__`**(*arg*)

`ScaleDegree.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ScaleDegree.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`ScaleDegree.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ScaleDegree.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

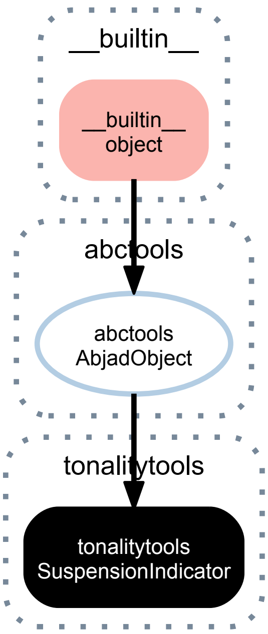
Raise exception.

`ScaleDegree.`**`__ne__`**(*arg*)

`ScaleDegree.`**`__repr__`**()

`ScaleDegree.`**`__str__`**()

42.1.10 tonalitytools.SuspensionIndicator



class tonalitytools.**SuspensionIndicator**(**args*)

New in version 2.0. Indicator of 9-8, 7-6, 4-3, 2-1 and other types of suspension typical of, for example, the Bach chorales.

Value object that can not be changed after instantiation.

Read-only properties

SuspensionIndicator.**chord_name**

SuspensionIndicator.**figured_bass_pair**

SuspensionIndicator.**figured_bass_string**

SuspensionIndicator.**is_empty**

SuspensionIndicator.**start**

SuspensionIndicator.**stop**

SuspensionIndicator.**storage_format**

Storage format of Abjad object.

Return string.

SuspensionIndicator.**title_string**

Special methods

SuspensionIndicator.**__eq__**(*arg*)

SuspensionIndicator.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SuspensionIndicator.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

SuspensionIndicator.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SuspensionIndicator.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

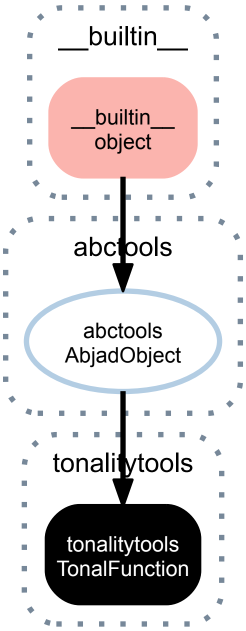
Raise exception.

SuspensionIndicator.**__ne__**(*arg*)

SuspensionIndicator.**__repr__**()

SuspensionIndicator.**__str__**()

42.1.11 tonalitytools.TonalFunction



class tonalitytools.**TonalFunction**(**args*)

New in version 2.0. Abjad model of functions in tonal harmony: I, I6, I64, V, V7, V43, V42, bII, bII6, etc., also i, i6, i64, v, v7, etc.

Value object that can not be changed after instantiation.

Read-only properties

TonalFunction.**bass_scale_degree**

TonalFunction.**extent**

TonalFunction.**figured_bass_string**

TonalFunction.**inversion**

TonalFunction.**markup**

TonalFunction.**quality**

TonalFunction.**root_scale_degree**

TonalFunction.**scale_degree**

TonalFunction.**storage_format**

Storage format of Abjad object.

Return string.

TonalFunction.**suspension**

TonalFunction.**symbolic_string**

Special methods

TonalFunction.**__eq__**(*arg*)

TonalFunction.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

TonalFunction.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

TonalFunction.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TonalFunction.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

TonalFunction.**__ne__**(*arg*)

TonalFunction.**__repr__**()

42.2 Functions

42.2.1 tonalitytools.analyze_chord

tonalitytools.**analyze_chord**(*expr*)
New in version 2.0. Analyze *expr* and return chord class.

```
>>> chord = Chord([7, 10, 12, 16], (1, 4))
>>> tonalitytools.analyze_chord(chord)
CDominantSeventhInSecondInversion
```

Return none when no tonal chord is understood.

```
>>> chord = Chord(['c', 'cs', 'd'], (1, 4))
>>> tonalitytools.analyze_chord(chord) is None
True
```

Raise tonal harmony error when chord can not analyze.

42.2.2 tonalitytools.analyze_incomplete_chord

tonalitytools.**analyze_incomplete_chord**(*expr*)
New in version 2.0. Analyze *expr* and return chord class based on incomplete pitches.

```
>>> tonalitytools.analyze_incomplete_chord(Chord([7, 11], (1, 4)))
GMajorTriadInRootPosition
```

```
>>> tonalitytools.analyze_incomplete_chord(Chord(['fs', 'g', 'b'], (1, 4)))
GMajorSeventhInSecondInversion
```

Return chord class.

42.2.3 tonalitytools.analyze_incomplete_tonal_function

tonalitytools.**analyze_incomplete_tonal_function**(*expr*, *key_signature*)
New in version 2.0. Analyze tonal function of *expr* according to *key_signature*:

```
>>> chord = Chord("<c' e'>4")
>>> key_signature = contexttools.KeySignatureMark('g', 'major')
>>> tonalitytools.analyze_incomplete_tonal_function(chord, key_signature)
IVMajorTriadInRootPosition
```

Return tonal function.

42.2.4 tonalitytools.analyze_tonal_function

tonalitytools.**analyze_tonal_function**(*expr*, *key_signature*)
New in version 2.0. Analyze *expr* and return tonal function according to *key_signature*.

```
>>> chord = Chord(['ef', 'g', 'bf'], (1, 4))
>>> key_signature = contexttools.KeySignatureMark('c', 'major')
>>> tonalitytools.analyze_tonal_function(chord, key_signature)
FlatIIIMajorTriadInRootPosition
```

Return none when no tonal function is understood.

```
>>> chord = Chord(['c', 'cs', 'd'], (1, 4))
>>> key_signature = contexttools.KeySignatureMark('c', 'major')
>>> tonalitytools.analyze_tonal_function(chord, key_signature) is None
True
```

Return tonal function or none.

42.2.5 tonalitytools.are_scalar_notes

tonalitytools.**are_scalar_notes**(**expr*)
New in version 2.0. True when notes in *expr* are scalar:

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> tonalitytools.are_scalar_notes(t[:])
True
```

Otherwise false:

```
>>> tonalitytools.are_scalar_notes(Note("c'4"), Note("c'4"))
False
```

Return boolean.

42.2.6 tonalitytools.are_stepwise_ascending_notes

tonalitytools.**are_stepwise_ascending_notes**(**expr*)
New in version 2.0. True when notes in *expr* are stepwise ascending:

```
>>> staff = Staff("c'16 d' e' f' g'2")
>>> time_signature = contexttools.TimeSignatureMark((3, 4))(staff)
```

```
>>> show(staff)
```

```
>>> tonalitytools.are_stepwise_ascending_notes(staff)
True
```

Otherwise false:

```
>>> staff[:4] = "f'16 e' d' c'"
```

```
>>> show(staff)
```

```
>>> tonalitytools.are_stepwise_ascending_notes(staff)
False
```

Return boolean.

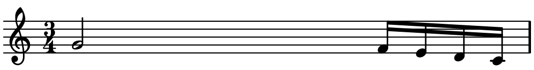
42.2.7 tonalitytools.are_stepwise_descending_notes

tonalitytools.**are_stepwise_descending_notes**(**expr*)

New in version 2.0. True when notes in *expr* are stepwise descending:

```
>>> staff = Staff("g'2 f'16 e' d' c'")
>>> time_signature = contexttools.TimeSignatureMark((3, 4))(staff)
```

```
>>> show(staff)
```



```
>>> tonalitytools.are_stepwise_descending_notes(staff)
True
```

Otherwise false:

```
>>> staff[-4:] = "c'16 d' e' f'"
```

```
>>> show(staff)
```

```
>>> tonalitytools.are_stepwise_descending_notes(staff)
False
```

Return boolean.

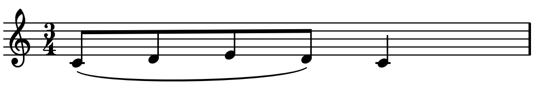
42.2.8 tonalitytools.are_stepwise_notes

tonalitytools.**are_stepwise_notes**(**expr*)

New in version 2.0. True when notes in *expr* are stepwise:

```
>>> staff = Staff("c'8 ( d' e' d' ) c'4")
>>> time_signature = contexttools.TimeSignatureMark((3, 4))(staff)
```

```
>>> show(staff)
```

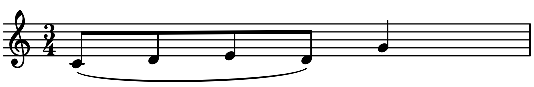


```
>>> tonalitytools.are_stepwise_notes(staff)
True
```

Otherwise false:

```
>>> staff[-1] = "g'4"
```

```
>>> show(staff)
```



```
>>> tonalitytools.are_stepwise_notes(staff)
False
```

Return boolean.

42.2.9 tonalitytools.chord_class_cardinality_to_extent

tonalitytools.**chord_class_cardinality_to_extent**(*cardinality*)
..versionadded:: 2.0

Change integer chord class *cardinality* to integer chord class extent:

```
>>> tonalitytools.chord_class_cardinality_to_extent(4)
7
```

The function above indicates that a tertian chord with 4 unique pitches qualifies as a seventh chord.

42.2.10 tonalitytools.chord_class_extent_to_cardinality

tonalitytools.**chord_class_extent_to_cardinality**(*extent*)
..versionadded:: 2.0

Change integer chord class *extent* to integer chord class cardinality:

```
>>> tonalitytools.chord_class_extent_to_cardinality(7)
4
```

The call above shows that a seventh chord comprises 4 unique pitch-classes.

42.2.11 tonalitytools.chord_class_extent_to_extent_name

tonalitytools.**chord_class_extent_to_extent_name**(*extent*)
New in version 2.0. Change integer chord class *extent* to extent name.

```
>>> tonalitytools.chord_class_extent_to_extent_name(7)
'seventh'
```

The call above shows that a tertian chord subtending 7 staff spaces qualifies as a seventh chord.

42.2.12 tonalitytools.diatonic_interval_class_segment_to_chord_quality_string

tonalitytools.**diatonic_interval_class_segment_to_chord_quality_string**(*dic_seg*)
New in version 2.0. Change diatonic interval-class segment *dic_seg* to chord quality string:

```
>>> dic_seg = pitchtools.InversionEquivalentDiatonicIntervalClassSegment([
...   pitchtools.InversionEquivalentDiatonicIntervalClass('major', 3),
...   pitchtools.InversionEquivalentDiatonicIntervalClass('minor', 3),])
>>> tonalitytools.diatonic_interval_class_segment_to_chord_quality_string(dic_seg)
'major'
```

Todo

Implement `diatonic_interval_class_set_to_chord_quality_string()`.

42.2.13 tonalitytools.is_neighbor_note

tonalitytools.**is_neighbor_note**(*note*)
New in version 2.0. True when *note* is preceeded by a stepwise interval in one direction and followed by a stepwise interval in the other direction. Otherwise false.

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> for note in t:
...     print '%s\t%s' % (note, tonalitytools.is_neighbor_note(note))
...
c'8     False
```

```
d'8      False
e'8      False
f'8      False
```

Return boolean.

42.2.14 tonalitytools.is_passing_tone

tonalitytools.**is_passing_tone**(*note*)
New in version 2.0. True when *note* is both preceeded and followed by scalewise sibling notes. Otherwise false.

```
>>> t = Staff("c'8 d'8 e'8 f'8")
>>> for note in t:
...     print '%s\t%s' % (note, tonalitytools.is_passing_tone(note))
...
c'8      False
d'8      True
e'8      True
f'8      False
```

Return boolean.

42.2.15 tonalitytools.is_unlikely_melodic_diatonic_interval_in_chorale

tonalitytools.**is_unlikely_melodic_diatonic_interval_in_chorale**(*mdi*)
New in version 2.0. True when *mdi* is unlikely melodic diatonic interval in JSB chorale.

```
>>> mdi = pitchtools.MelodicDiatonicInterval('major', 7)
>>> tonalitytools.is_unlikely_melodic_diatonic_interval_in_chorale(mdi)
True
```

Otherwise False.

```
>>> mdi = pitchtools.MelodicDiatonicInterval('major', 2)
>>> tonalitytools.is_unlikely_melodic_diatonic_interval_in_chorale(mdi)
False
```

Return boolean.

42.2.16 tonalitytools.make_all_notes_in_ascending_and_descending_diatonic_scale

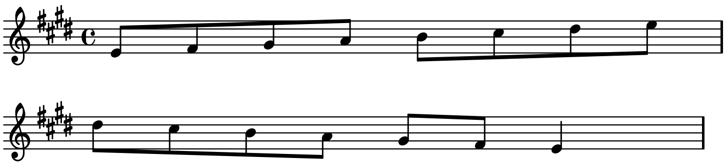
tonalitytools.**make_all_notes_in_ascending_and_descending_diatonic_scale**(*key_signature=None*)
New in version 2.0. Make all notes in ascending and descending diatonic scale of *key_signature*:

```
>>> score = tonalitytools.make_all_notes_in_ascending_and_descending_diatonic_scale(
... contexttools.KeySignatureMark('E', 'major'))
```

```
>>> f(score)
\new Score \with {
    tempoWholesPerMinute = #(ly:make-moment 30 1)
} <<
    \new Staff {
        \key e \major
        e'8
        fs'8
        gs'8
        a'8
        b'8
        cs''8
        ds''8
        e''8
        ds''8
        cs''8
```

```
            b'8
            a'8
            gs'8
            fs'8
            e'4
        }
>>
```

```
>>> show(score)
```



Return score suitable for MIDI playback.

42.2.17 tonalitytools.make_first_n_notes_in_ascending_diatonic_scale

`tonalitytools.make_first_n_notes_in_ascending_diatonic_scale`(*count*, *written_duration=None*, *key_signature=None*)

Make first *count* notes in ascending diatonic scale according to *key_signature*. Set *written_duration* equal to *written_duration* or `1/8`:

```
>>> notes = tonalitytools.make_first_n_notes_in_ascending_diatonic_scale(8)
>>> staff = Staff(notes)
```

```
>>> f(staff)
\new Staff {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
    b'8
    c''8
}
```

```
>>> show(staff)
```

Allow nonassignable *written_duration*:

```
>>> notes = tonalitytools.make_first_n_notes_in_ascending_diatonic_scale(
...     4, Duration(5, 16))
>>> staff = Staff(notes)
>>> time_signature = contexttools.TimeSignatureMark((5, 4))(staff)
```

```
>>> f(staff)
\new Staff {
    \time 5/4
    c'4 ~
    c'16
    d'4 ~
    d'16
    e'4 ~
    e'16
    f'4 ~
    f'16
}
```

```
>>> show(staff)
```

Return list of notes.

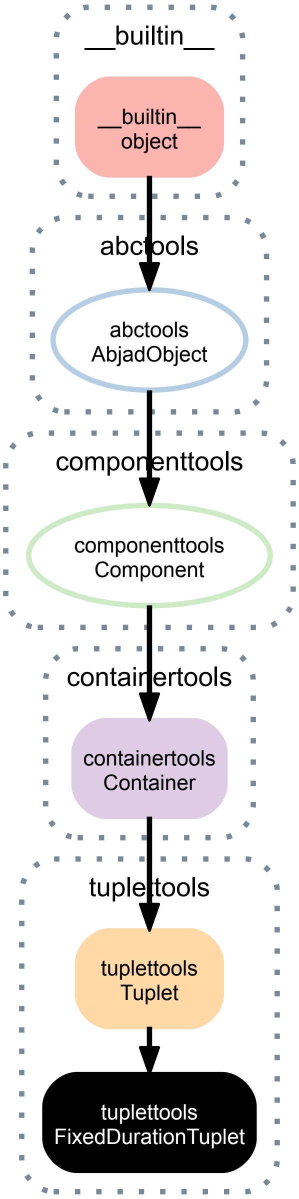
CHAPTER

FORTYTHREE

TUPLETTOOLS

43.1 Concrete Classes

43.1.1 tuplettools.FixedDurationTuplet



class tuplettools.**FixedDurationTuplet** (*duration*, *music=None*, ***kwargs*)

Abjad tuplet of fixed duration and variable multiplier:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Fraction(2, 8), "c'8 d'8 e'8")
```

```
>>> tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> show(tuplet)
```


```
>>> tuplet.append("fs'4")
>>> f(tuplet)
\times 2/5 {
    c'8
    d'8
    e'8
    fs'4
}
```

```
>>> show(tuplet)
```

Return fixed-duration tuplet.

Read-only properties

FixedDurationTuplet.**contents_duration**

FixedDurationTuplet.**descendants**

Read-only reference to component descendants score selection.

FixedDurationTuplet.**duration**

FixedDurationTuplet.**duration_in_seconds**

FixedDurationTuplet.**has_non_power_of_two_denominator**

Read-only boolean true when multiplier numerator is not power of two. Otherwise false:

```
>>> tuplet = Tuplet((3, 5), "c'8 d'8 e'8 f'8 g'8")
>>> tuplet.has_non_power_of_two_denominator
True
```

Return boolean.

FixedDurationTuplet.**has_power_of_two_denominator**

Read-only boolean true when multiplier numerator is power of two. Otherwise false:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.has_power_of_two_denominator
True
```

Return boolean.

FixedDurationTuplet.**implied_prolation**

Tuplet implied prolation.

Defined equal to tuplet multiplier.

Return multiplier.

FixedDurationTuplet.**is_augmentation**

True when multiplier is greater than 1. Otherwise false:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.is_augmentation
False
```

Return boolean.

FixedDurationTuplet.**is_diminution**

True when multiplier is less than 1. Otherwise false:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.is_diminution
True
```

Return boolean.

FixedDurationTuplet.**is_trivial**

True when tuplet multiplier is one. Otherwise false:

```
>>> tuplet = Tuplet((1, 1), "c'8 d'8 e'8")
>>> tuplet.is_trivial
True
```

Return boolean.

FixedDurationTuplet.**leaves**

Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

FixedDurationTuplet.**lilypond_format**

FixedDurationTuplet.**lineage**

Read-only reference to component lineage score selection.

FixedDurationTuplet.**multiplied_duration**

Read-only multiplied duration of tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Return duration.

FixedDurationTuplet.**music**

Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

FixedDurationTuplet.**override**

Read-only reference to LilyPond grob override component plug-in.

FixedDurationTuplet.**parent**

FixedDurationTuplet.**parentage**
Read-only reference to component parentage score selection.

FixedDurationTuplet.**preprolated_duration**
Duration prior to prolation:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.preprolated_duration
Duration(1, 4)
```

Return duration.

FixedDurationTuplet.**prolation**

FixedDurationTuplet.**ratio_string**
Read-only tuplet multiplier formatted with colon as ratio:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.ratio_string
'3:2'
```

Return string.

FixedDurationTuplet.**set**
Read-only reference LilyPond context setting component plug-in.

FixedDurationTuplet.**spanners**
Read-only reference to unordered set of spanners attached to component.

FixedDurationTuplet.**storage_format**
Storage format of Abjad object.

Return string.

FixedDurationTuplet.**timespan**
Read-only timespan of component.

FixedDurationTuplet.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

FixedDurationTuplet.**force_fraction**
Read / write boolean to force `n:m` fraction in LilyPond format:

```
>>> tuplet = Tuplet(Fraction(2, 3), "c'8 d'8 e'8")
```

```
>>> tuplet.force_fraction is None
True
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> tuplet.force_fraction = True
```

```
>>> f(tuplet)
\fraction \times 2/3 {
    c'8
    d'8
    e'8
}
```

Return boolean or none.

`FixedDurationTuplet.`**`is_invisible`**

Read / write boolean to output LilyPond `\scaledDurations` instead of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> tuplet.is_invisible = True
```

```
\scaleDurations #'(2 . 3) {
    c'8
    d'8
    e'8
}
```

This has the effect of rendering no no tuplet bracket and no tuplet number while preserving the rhythmic value of the tuplet and the contents of the tuplet.

Return boolean or none.

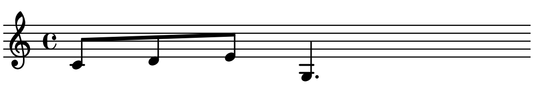
`FixedDurationTuplet.`**`is_parallel`**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

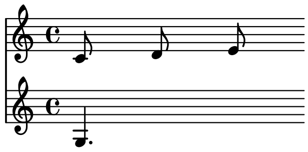
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

FixedDurationTuplet.**multiplier**
Read-only multiplier of tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> tuplet.multiplier
Multiplier(2, 3)
```

Return multiplier.

FixedDurationTuplet.**preferred_denominator**
New in version 2.0. Integer denominator in terms of which tuplet fraction should format:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.preferred_denominator = 4
```

```
>>> f(tuplet)
\times 4/6 {
    c'8
    d'8
    e'8
}
```

Return positive integer or none.

FixedDurationTuplet.**target_duration**
Read / write target duration of fixed-duration tuplet:

```
>>> tuplet = tuplettools.FixedDurationTuplet((1, 4), "c'8 d'8 e'8")
>>> tuplet.target_duration
Duration(1, 4)
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> tuplet.target_duration = Duration(5, 8)
>>> f(tuplet)
\fraction \times 5/3 {
    c'8
    d'8
    e'8
}
```

Return duration.

Methods

FixedDurationTuplet.**append**(*component*)
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```

Return none.

`FixedDurationTuplet.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```

Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`FixedDurationTuplet.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`FixedDurationTuplet.`**insert** (*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

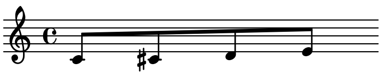
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`FixedDurationTuplet.`**pop** (*i=-1*)

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

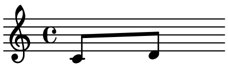
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

FixedDurationTuplet.**remove**(*component*)

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

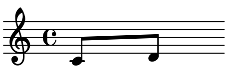
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

FixedDurationTuplet.**trim**(*start*, *stop='unused'*)

Trim fixed-duration tuplet elements from *start* to *stop*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Fraction(2, 8), "c'8 d'8 e'8")
>>> tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> tuplet.trim(2)
>>> tuplet
FixedDurationTuplet(1/6, [c'8, d'8])
```

Preserve fixed-duration tuplet multiplier.

Adjust fixed-duration tuplet duration.

Return none.

Special methods

FixedDurationTuplet.**__add__**(*arg*)
Add two tuplets of same type and with same multiplier.

FixedDurationTuplet.**__contains__**(*expr*)
True if expr is in container, otherwise False.

FixedDurationTuplet.**__copy__**(**args*)

FixedDurationTuplet.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

FixedDurationTuplet.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

FixedDurationTuplet.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationTuplet.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

FixedDurationTuplet.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

FixedDurationTuplet.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

FixedDurationTuplet.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

FixedDurationTuplet.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationTuplet.**__len__**()
Return nonnegative integer number of components in container.

FixedDurationTuplet.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

FixedDurationTuplet.**__mul__**(*n*)

FixedDurationTuplet.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

FixedDurationTuplet.**__radd__**(*expr*)
Extend container by contents of expr to the right.

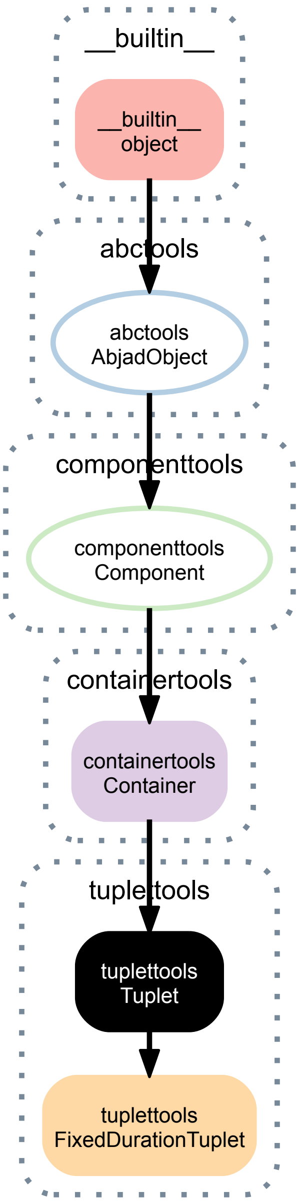
FixedDurationTuplet.**__repr__**()

FixedDurationTuplet.**__rmul__**(*n*)

FixedDurationTuplet.**__setitem__**(*i, expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

FixedDurationTuplet.**__str__**()

43.1.2 tuplettools.Tuplet



class `tuplettools.`**`Tuplet`** (*multiplier*, *music=None*, ***kwargs*)

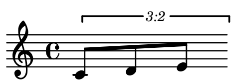
Abjad model of a tuplet:

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'8 d'8 e'8")
```

```
>>> tuplet
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> show(tuplet)
```

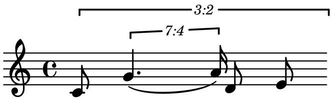


Tuplets, like any counttime container, may nest arbitrarily:

```
>>> second_tuplet = Tuplet((4, 7), "g'4. ( a'16 )")
>>> tuplet.insert(1, second_tuplet)
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    \times 4/7 {
        g'4. (
        a'16 )
    }
    d'8
    e'8
}
```

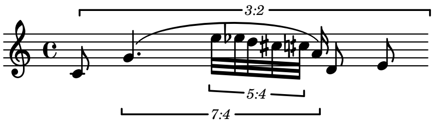
```
>>> show(tuplet)
```




```
>>> third_tuplet = Tuplet((4, 5), "e''32 [ ef''32 d''32 cs''32 cqs''32 ]")
>>> second_tuplet.insert(1, third_tuplet)
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    \times 4/7 {
        g'4. (
        \times 4/5 {
            e''32 [
            ef''32
            d''32
            cs''32
            cqs''32 ]
        }
        a'16 )
    }
    d'8
    e'8
}
```

```
>>> show(tuplet)
```



Return tuplet object.

Read-only properties

`Tuplet.`**`contents_duration`**

`Tuplet.`**`descendants`**
Read-only reference to component descendants score selection.

`Tuplet.`**`duration`**

`Tuplet.`**`duration_in_seconds`**

`Tuplet.`**`has_non_power_of_two_denominator`**
Read-only boolean true when multiplier numerator is not power of two. Otherwise false:

```
>>> tuplet = Tuplet((3, 5), "c'8 d'8 e'8 f'8 g'8")
>>> tuplet.has_non_power_of_two_denominator
True
```

Return boolean.

`Tuplet.`**`has_power_of_two_denominator`**
Read-only boolean true when multiplier numerator is power of two. Otherwise false:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.has_power_of_two_denominator
True
```

Return boolean.

`Tuplet.`**`implied_prolation`**
Tuplet implied prolation.

Defined equal to tuplet multiplier.

Return multiplier.

`Tuplet.`**`is_augmentation`**
True when multiplier is greater than 1. Otherwise false:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.is_augmentation
False
```

Return boolean.

`Tuplet.`**`is_diminution`**
True when multiplier is less than 1. Otherwise false:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.is_diminution
True
```

Return boolean.

`Tuplet.`**`is_trivial`**
True when tuplet multiplier is one. Otherwise false:

```
>>> tuplet = Tuplet((1, 1), "c'8 d'8 e'8")
>>> tuplet.is_trivial
True
```

Return boolean.

`Tuplet.`**`leaves`**
Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

`Tuplet.`**`lilypond_format`**

`Tuplet.`**`lineage`**
Read-only reference to component lineage score selection.

`Tuplet.`**`multiplied_duration`**
Read-only multiplied duration of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.multiplied_duration
Duration(1, 4)
```

Return duration.

Tuplet.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

Tuplet.**override**
Read-only reference to LilyPond grob override component plug-in.

Tuplet.**parent**

Tuplet.**parentage**
Read-only reference to component parentage score selection.

Tuplet.**preprolated_duration**
Duration prior to prolation:

```
>>> t = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> t.preprolated_duration
Duration(1, 4)
```

Return duration.

Tuplet.**prolation**

Tuplet.**ratio_string**
Read-only tuplet multiplier formatted with colon as ratio:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.ratio_string
'3:2'
```

Return string.

Tuplet.**set**
Read-only reference LilyPond context setting component plug-in.

Tuplet.**spanners**
Read-only reference to unordered set of spanners attached to component.

Tuplet.**storage_format**
Storage format of Abjad object.

Return string.

Tuplet.**timespan**
Read-only timespan of component.

Tuplet.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

Tuplet.**force_fraction**
Read / write boolean to force `n:m` fraction in LilyPond format:

```
>>> tuplet = Tuplet(Fraction(2, 3), "c'8 d'8 e'8")
```

```
>>> tuplet.force_fraction is None
True
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> tuplet.force_fraction = True
```

```
>>> f(tuplet)
\fraction \times 2/3 {
    c'8
    d'8
    e'8
}
```

Return boolean or none.

`Tuplet.`**`is_invisible`**

Read / write boolean to output LilyPond `\scaledDurations` instead of tuplet:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
```

```
>>> f(tuplet)
\times 2/3 {
    c'8
    d'8
    e'8
}
```

```
>>> tuplet.is_invisible = True
```

```
\scaleDurations #'(2 . 3) {
    c'8
    d'8
    e'8
}
```

This has the effect of rendering no no tuplet bracket and no tuplet number while preserving the rhythmic value of the tuplet and the contents of the tuplet.

Return boolean or none.

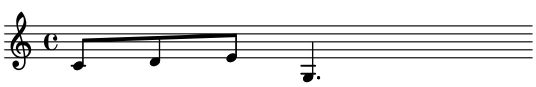
`Tuplet.`**`is_parallel`**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

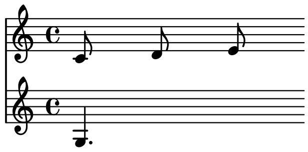
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

Tuplet.**multiplier**
Read / write tuplet multiplier:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.multiplier
Multiplier(2, 3)
```

Return multiplier.

Tuplet.**preferred_denominator**
New in version 2.0. Integer denominator in terms of which tuplet fraction should format:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
>>> tuplet.preferred_denominator = 4
```

```
>>> f(tuplet)
\times 4/6 {
    c'8
    d'8
    e'8
}
```

Return positive integer or none.

Methods

Tuplet.**append**(*component*)
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

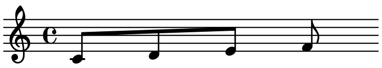
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`Tuplet.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

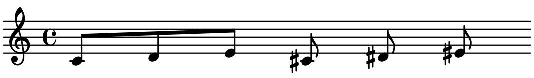
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Tuplet.`**`index`**`(`*component*`)`

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Tuplet.insert(i, component)`
Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

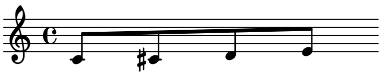
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Tuplet.pop(i=-1)`
Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

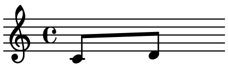
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Tuplet.`**`remove`**`(`*component*`)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

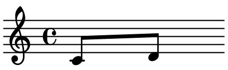
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

`Tuplet.`**`__add__`**`(`*arg*`)`

Add two tuplets of same type and with same multiplier.

`Tuplet.`**`__contains__`**`(`*expr*`)`

True if expr is in container, otherwise False.

`Tuplet.`**`__copy__`**`(`**args*`)`

`Tuplet.`**`__delitem__`**`(`*i*`)`

Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

`Tuplet.`**`__eq__`**`(`*expr*`)`

True when `id(self)` equals `id(expr)`.

Return boolean.

`Tuplet.`**`__ge__`**`(`*expr*`)`

Abjad objects by default do not implement this method.

Raise exception.

Tuplet.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

Tuplet.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Tuplet.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

Tuplet.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

Tuplet.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Tuplet.**__len__**()
Return nonnegative integer number of components in container.

Tuplet.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Tuplet.**__mul__**(*n*)

Tuplet.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Tuplet.**__radd__**(*expr*)
Extend container by contents of expr to the right.

Tuplet.**__repr__**()

Tuplet.**__rmul__**(*n*)

Tuplet.**__setitem__**(*i*, *expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

Tuplet.**__str__**()

43.2 Functions

43.2.1 tuplettools.all_are_tuplets

tuplettools.**all_are_tuplets**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad tuplets:

```
>>> tuplet = Tuplet((2, 3), "c'8 d'8 e'8")
```

```
>>> tuplettools.all_are_tuplets([tuplet])
True
```

True when *expr* is an empty sequence:

```
>>> tuplettools.all_are_tuplets([])
True
```

Otherwise false:

```
>>> tuplettools.all_are_tuplets('foo')
False
```

Function wraps `componenttools.all_are_components()`.

Return boolean.

43.2.2 tuplettools.change_augmented_tuplets_in_expr_to_diminished

tuplettools.**change_augmented_tuplets_in_expr_to_diminished**(*tuplet*)

New in version 2.0. Multiply the written duration of the leaves in *tuplet* by the least power of 2 necessary to diminshed *tuplet*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 4), "c'8 d'8 e'8")
```

```
>>> tuplet
FixedDurationTuplet(1/2, [c'8, d'8, e'8])
```

```
>>> tuplettools.change_augmented_tuplets_in_expr_to_diminished(tuplet)
FixedDurationTuplet(1/2, [c'4, d'4, e'4])
```

Note: Does not yet work with nested tuplets.

Return *tuplet*.

43.2.3 tuplettools.change_diminished_tuplets_in_expr_to_augmented

tuplettools.**change_diminished_tuplets_in_expr_to_augmented**(*tuplet*)

New in version 2.0. Divide the written duration of the leaves in *tuplet* by the least power of 2 necessary to augment *tuplet*:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
```

```
>>> tuplet
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> tuplettools.change_diminished_tuplets_in_expr_to_augmented(tuplet)
FixedDurationTuplet(1/4, [c'16, d'16, e'16])
```

Note: Does not yet work with nested tuplets.

Return *tuplet*.

43.2.4 tuplettools.change_fixed_duration_tuplets_in_expr_to_tuplets

tuplettools.**change_fixed_duration_tuplets_in_expr_to_tuplets**(*expr*)

New in version 2.9. Change fixed-duration tuplets in *expr* to tuplets:

```
>>> staff = Staff(2 * tuplettools.FixedDurationTuplet((2, 8), "c'8 d'8 e'8"))
```

```
>>> for x in staff[:]:
...     x
...
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

```
>>> tuplettools.change_fixed_duration_tuplets_in_expr_to_tuplets(staff)
[Tuplet(2/3, [c'8, d'8, e'8]), Tuplet(2/3, [c'8, d'8, e'8])]
```

```
>>> staff[:]
Selection(Tuplet(2/3, [c'8, d'8, e'8]), Tuplet(2/3, [c'8, d'8, e'8]))
```

Return tuplets.

43.2.5 tuplettools.change_tuplets_in_expr_to_fixed_duration_tuplets

tuplettools.**change_tuplets_in_expr_to_fixed_duration_tuplets**(*expr*)
New in version 2.9. Change tuplets in *expr* to fixed-duration tuplets:

```
>>> staff = Staff(r"\times 2/3 { c'8 d'8 e'8 } \times 2/3 { c'8 d'8 e'8 }")
```

```
>>> staff[:]
Selection(Tuplet(2/3, [c'8, d'8, e'8]), Tuplet(2/3, [c'8, d'8, e'8]))
```

```
>>> tuplettools.change_tuplets_in_expr_to_fixed_duration_tuplets(staff)
[FixedDurationTuplet(1/4, [c'8, d'8, e'8]), FixedDurationTuplet(1/4, [c'8, d'8, e'8])]
```

```
>>> for x in staff[:]:
...     x
...
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

Return tuplets.

43.2.6 tuplettools.fix_contents_of_tuplets_in_expr

tuplettools.**fix_contents_of_tuplets_in_expr**(*tuplet*)
Scale *tuplet* contents by power of two if tuplet multiplier less than 1/2 or greater than 2. Return tuplet.

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'4 d'4 e'4")
```

```
>>> tuplet
FixedDurationTuplet(1/4, [c'4, d'4, e'4])
```

```
>>> tuplettools.fix_contents_of_tuplets_in_expr(tuplet)
FixedDurationTuplet(1/4, [c'8, d'8, e'8])
```

Return *tuplet*.

43.2.7 tuplettools.fuse_tuplets

tuplettools.**fuse_tuplets**(*tuplets*)
Fuse parent-contiguous *tuplets*:

```
>>> t1 = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 d'8 e'8")
>>> beamtools.BeamSpanner(t1[:])
BeamSpanner(c'8, d'8, e'8)
>>> t2 = tuplettools.FixedDurationTuplet(Duration(2, 16), "c'16 d'16 e'16")
>>> spannertools.SlurSpanner(t2[:])
SlurSpanner(c'16, d'16, e'16)
>>> staff = Staff([t1, t2])
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
    }
    \times 2/3 {
        c'16 (
        d'16
```

```
            e'16 )
        }
    }
```

```
>>> tuplettools.fuse_tuplets(staff[:])
FixedDurationTuplet(3/8, [c'8, d'8, e'8, c'16, d'16, e'16])
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8 [
        d'8
        e'8 ]
        c'16 (
        d'16
        e'16 )
    }
}
```

Return new tuplet.

Fuse zero or more parent-contiguous *tuplets*.

Allow in-score *tuplets*.

Allow outside-of-score *tuplets*.

All *tuplets* must carry the same multiplier.

All *tuplets* must be of the same type.

43.2.8 tuplettools.get_first_tuplet_in_improper_parentage_of_component

tuplettools.**get_first_tuplet_in_improper_parentage_of_component**(*component*)
New in version 2.0. Get first tuplet in improper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> Tuplet(Fraction(2, 3), staff[:3])
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    f'8
}
```

```
>>> tuplettools.get_first_tuplet_in_improper_parentage_of_component(staff.leaves[1])
Tuplet(2/3, [c'8, d'8, e'8])
```

Return tuplet or none.

43.2.9 tuplettools.get_first_tuplet_in_proper_parentage_of_component

tuplettools.**get_first_tuplet_in_proper_parentage_of_component**(*component*)
New in version 2.0. Get first tuplet in proper parentage of *component*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> Tuplet(Fraction(2, 3), staff[:3])
Tuplet(2/3, [c'8, d'8, e'8])
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    f'8
}
```

```
>>> tuplettools.get_first_tuplet_in_proper_parentage_of_component(staff.leaves[1])
Tuplet(2/3, [c'8, d'8, e'8])
```

Return tuplet or none.

43.2.10 tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration

`tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration`(*leaf*, *n*, *is_diminution=True*)

Change *leaf* to tuplet *n* notes of equal written duration.

Example 1. Change leaf to augmented tuplet:

```
>>> for n in range(1, 11):
...     note = Note(0, (3, 16))
...     tuplet = tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration(
...         note, n, is_diminution=False)
...     print tuplet
...
{@ 1:1 c'8. @}
{@ 1:1 c'16., c'16. @}
{@ 1:1 c'16, c'16, c'16 @}
{@ 1:1 c'32., c'32., c'32., c'32. @}
{@ 5:8 c'64., c'64., c'64., c'64., c'64. @}
{@ 1:1 c'32, c'32, c'32, c'32, c'32, c'32 @}
{@ 7:8 c'64., c'64., c'64., c'64., c'64., c'64., c'64. @}
{@ 1:1 c'64., c'64., c'64., c'64., c'64., c'64., c'64., c'64. @}
{@ 3:4 c'64, c'64, c'64, c'64, c'64, c'64, c'64, c'64, c'64 @}
{@ 5:8 c'128., c'128., c'128., c'128., c'128., c'128., c'128., c'128., c'128., c'128. @}
```

Example 2. Change to leaf diminished tuplet:

```
>>> for n in range(1, 11):
...     note = Note(0, (3, 16))
...     tuplet = tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration(
...         note, n, is_diminution=True)
...     print tuplet
...
{@ 1:1 c'8. @}
{@ 1:1 c'16., c'16. @}
{@ 1:1 c'16, c'16, c'16 @}
{@ 1:1 c'32., c'32., c'32., c'32. @}
{@ 5:4 c'32., c'32., c'32., c'32., c'32. @}
{@ 1:1 c'32, c'32, c'32, c'32, c'32, c'32 @}
{@ 7:4 c'32., c'32., c'32., c'32., c'32., c'32., c'32. @}
{@ 1:1 c'64., c'64., c'64., c'64., c'64., c'64., c'64., c'64. @}
{@ 3:2 c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32, c'32 @}
{@ 5:4 c'64., c'64., c'64., c'64., c'64., c'64., c'64., c'64., c'64., c'64. @}
```

Return fixed-duration tuplet.

43.2.11 tuplettools.leaf_to_tuplet_with_ratio

`tuplettools.leaf_to_tuplet_with_ratio`(*leaf*, *proportions*, *is_diminution=True*)

Change *leaf* to tuplet with *proportions*.

Example 1. Change *leaf* to augmented tuplet with *proportions*:

```
>>> note = Note(0, (3, 16))
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1], is_diminution=False)
{@ 1:1 c'8. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2], is_diminution=False)
{@ 1:1 c'16, c'8 @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2], is_diminution=False)
{@ 5:8 c'64., c'32., c'32. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3], is_diminution=False)
{@ 2:3 c'64, c'32, c'32, c'32. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3, 3], is_diminution=False)
{@ 11:12 c'64, c'32, c'32, c'32., c'32. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3, 3, 4], is_diminution=False)
{@ 5:8 c'128, c'64, c'64, c'64., c'64., c'32 @}
```

Example 2. Change *leaf* to diminished tuplet:

```
>>> note = Note(0, (3, 16))
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1], is_diminution=True)
{@ 1:1 c'8. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2], is_diminution=True)
{@ 1:1 c'16, c'8 @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2], is_diminution=True)
{@ 5:4 c'32., c'16., c'16. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3], is_diminution=True)
{@ 4:3 c'32, c'16, c'16, c'16. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3, 3], is_diminution=True)
{@ 11:6 c'32, c'16, c'16, c'16., c'16. @}
```

```
>>> print tuplettools.leaf_to_tuplet_with_ratio(
...     note, [1, 2, 2, 3, 3, 4], is_diminution=True)
{@ 5:4 c'64, c'32, c'32, c'32., c'32., c'16 @}
```

Return fixed-duration tuplet.

43.2.12 tuplettools.make_tuplet_from_duration_and_ratio

`tuplettools.`**`make_tuplet_from_duration_and_ratio`**(*duration*, *proportions*, *avoid_dots=True*, *decrease_durations_monotonically=True*, *is_diminution=True*)

New in version 2.10. Make tuplet from *duration* and *proportions*.

Example set 1. Make augmented tuplet from *duration* and *proportions* and avoid dots.

Return tupletted leaves strictly without dots when all *proportions* equal `1`:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, 1, 1, -1, -1], avoid_dots=True,
...     is_diminution=False)
{@ 5:6 c'32, c'32, c'32, r32, r32 @}
```

Allow tupletted leaves to return with dots when some *proportions* do not equal `1`:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, -2, -2, 3, 3], avoid_dots=True,
...     is_diminution=False)
{@ 11:12 c'64, r32, r32, c'32., c'32. @}
```

Interpret nonassignable *proportions* according to *decrease_durations_monotonically*:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [5, -1, 5], avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=False)
{@ 11:12 c'64, c'16, r64, c'64, c'16 @}
```

Example set 2. Make augmented tuplet from *duration* and *proportions* and encourage dots:

```
>>> tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, 1, 1, -1, -1], avoid_dots=False,
...     is_diminution=False)
FixedDurationTuplet(3/16, [c'64., c'64., c'64., r64., r64.])
```

Interpret nonassignable *proportions* according to *decrease_durations_monotonically*:

```
>>> tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [5, -1, 5], avoid_dots=False,
...     decrease_durations_monotonically=False,
...     is_diminution=False)
FixedDurationTuplet(3/16, [c'32..., r128., c'32...])
```

Example set 3. Make diminished tuplet from *duration* and nonzero integer *proportions*.

Return tupletted leaves strictly without dots when all *proportions* equal `1`:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, 1, 1, -1, -1], avoid_dots=True,
...     is_diminution=True)
{@ 5:3 c'16, c'16, c'16, r16, r16 @}
```

Allow tupletted leaves to return with dots when some *proportions* do not equal `1`:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, -2, -2, 3, 3], avoid_dots=True,
...     is_diminution=True)
{@ 11:6 c'32, r16, r16, c'16., c'16. @}
```

Interpret nonassignable *proportions* according to *decrease_durations_monotonically*:

```
>>> print tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [5, -1, 5], avoid_dots=True,
...     decrease_durations_monotonically=False,
...     is_diminution=True)
{@ 11:6 c'32, c'8, r32, c'32, c'8 @}
```

Example set 4. Make diminished tuplet from *duration* and *proportions* and encourage dots:

```
>>> tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [1, 1, 1, -1, -1], avoid_dots=False,
...     is_diminution=True)
FixedDurationTuplet(3/16, [c'32., c'32., c'32., r32., r32.])
```

Interpret nonassignable *proportions* according to *direction*:

```
>>> tuplettools.make_tuplet_from_duration_and_ratio(
...     Duration(3, 16), [5, -1, 5], avoid_dots=False,
...     decrease_durations_monotonically=False,
```

```
...     is_diminution=True)
FixedDurationTuplet(3/16, [c'16..., r64., c'16...])
```

Reduce *proportions* relative to each other.

Interpret negative *proportions* as rests.

Return fixed-duration tuplet.

43.2.13 tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction

tuplettools.**make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction**(*proportions*, *(n, d)*)

Divide nonreduced fraction *(n, d)* according to *proportions*.

Return container when no prolation is necessary:

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1], (7, 16))
{c'4..}
```

Return fixed-duration tuplet when prolation is necessary:

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1, 2], (7, 16))
FixedDurationTuplet(7/16, [c'8, c'4])
```

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1, 2, 4], (7, 16))
FixedDurationTuplet(7/16, [c'16, c'8, c'4])
```

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1, 2, 4, 1], (7, 16))
FixedDurationTuplet(7/16, [c'16, c'8, c'4, c'16])
```

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1, 2, 4, 1, 2], (7, 16))
FixedDurationTuplet(7/16, [c'16, c'8, c'4, c'16, c'8])
```

```
>>> tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction(
...     [1, 2, 4, 1, 2, 4], (7, 16))
FixedDurationTuplet(7/16, [c'16, c'8, c'4, c'16, c'8, c'4])
```

Note: function interprets *d* as tuplet denominator.

Return tuplet or container.

43.2.14 tuplettools.move_prolation_of_tuplet_to_contents_of_tuplet_and_remove_tuplet

tuplettools.**move_prolation_of_tuplet_to_contents_of_tuplet_and_remove_tuplet**(*tuplet*)

Move prolation of *tuplet* to contents of *tuplet* and remove *tuplet*:

```
>>> t = Staff(r"\times 3/2 { c'8 [ d'8 } \times 3/2 { c'8 d'8 ] }")
```

```
>>> f(t)
\new Staff {
    \fraction \times 3/2 {
        c'8 [
        d'8
    }
    \fraction \times 3/2 {
        c'8
        d'8 ]
```

```
        }
    }
```

```
>>> tuplettools.move_prolation_of_tuplet_to_contents_of_tuplet_and_remove_tuplet(t[0])
Tuplet(3/2, [])
```

```
>>> f(t)
\new Staff {
    c'8. [
    d'8.
    \fraction \times 3/2 {
        c'8
        d'8 ]
    }
}
```

Return *tuplet*.

43.2.15 tuplettools.remove_trivial_tuplets_in_expr

tuplettools.**remove_trivial_tuplets_in_expr**(*expr*)

Remove trivial tuplets in *expr*:

```
>>> tuplet_1 = tuplettools.FixedDurationTuplet(Duration(1, 4), "c'8 d'8 e'8")
>>> tuplet_2 = tuplettools.FixedDurationTuplet(Duration(1, 4), "c'8 d'8")
>>> staff = Staff([tuplet_1, tuplet_2])
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    {
        c'8
        d'8
    }
}
```

```
>>> tuplettools.remove_trivial_tuplets_in_expr(staff)
```

```
>>> f(staff)
\new Staff {
    \times 2/3 {
        c'8
        d'8
        e'8
    }
    c'8
    d'8
}
```

Replace trivial tuplets with plain leaves.

Return none.

43.2.16 tuplettools.scale_contents_of_tuplets_in_expr_by_multiplier

tuplettools.**scale_contents_of_tuplets_in_expr_by_multiplier**(*tuplet*, *multiplier*)

Scale contents of fixed-duration *tuplet* by *multiplier*:

```
>>> tuplet = tuplettools.FixedDurationTuplet((3, 8), "c'8 d'8 e'8 f'8 g'8")
```

```
>>> f(tuplet)
\fraction \times 3/5 {
    c'8
    d'8
    e'8
    f'8
    g'8
}
```

```
>>> tuplettools.scale_contents_of_tuplets_in_expr_by_multiplier(tuplet, Fraction(2))
FixedDurationTuplet(3/4, [c'4, d'4, e'4, f'4, g'4])
```

```
>>> f(tuplet)
\fraction \times 3/5 {
    c'4
    d'4
    e'4
    f'4
    g'4
}
```

Preserve *tuplet* multiplier.

Return tuplet.

43.2.17 tuplettools.set_denominator_of_tuplets_in_expr_to_at_least

tuplettools.**set_denominator_of_tuplets_in_expr_to_at_least**(*expr*, *n*)

New in version 2.0. Set denominator of tuplets in *expr* to at least *n*:

```
>>> tuplet = Tuplet(Fraction(3, 5), "c'4 d'8 e'8 f'4 g'2")
```

```
>>> f(tuplet)
\fraction \times 3/5 {
    c'4
    d'8
    e'8
    f'4
    g'2
}
```

```
>>> tuplettools.set_denominator_of_tuplets_in_expr_to_at_least(tuplet, 8)
```

```
>>> f(tuplet)
\fraction \times 6/10 {
    c'4
    d'8
    e'8
    f'4
    g'2
}
```

Return none.

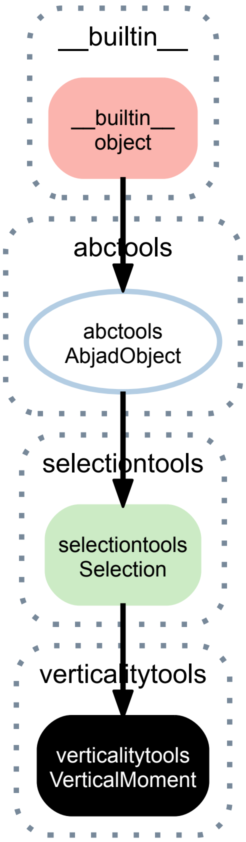
CHAPTER

FORTYFOUR

VERTICALITYTOOLS

44.1 Concrete Classes

44.1.1 verticalitytools.VerticalMoment



class `verticalitytools.`**`VerticalMoment`**(*offset*, *governors*, *components*)

Everything happening at a single moment in musical time:

```
>>> score = Score([])
>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("c'4 e'4 d'4 f'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" g2 f2"""))
>>> score.append(piano_staff)
```

```
f(score)
\new Score <<
    \new PianoStaff <<
        \new Staff {
```

```
                c'4
                e'4
                d'4
                f'4
            }
            \new Staff {
                \clef "bass"
                g2
                f2
            }
        >>
    >>
```

```
>>> for x in verticalitytools.iterate_vertical_moments_in_expr(score):
...     x
...
VerticalMoment(0, <<2>>)
VerticalMoment(1/4, <<2>>)
VerticalMoment(1/2, <<2>>)
VerticalMoment(3/4, <<2>>)
```

Create vertical moments with the getters and iterators implemented in the `verticalitytools` module.

Vertical moments are immutable.

Read-only properties

`VerticalMoment.`**`attack_count`**
Positive integer number of pitch carriers starting at vertical moment.

`VerticalMoment.`**`components`**
Read-only tuple of zero or more components happening at vertical moment.

It is always the case that `self.components = self.overlap_components + self.start_components`.

`VerticalMoment.`**`governors`**
Read-only tuple of one or more containers in which vertical moment is evaluated.

`VerticalMoment.`**`leaves`**
Read-only tuple of zero or more leaves at vertical moment.

`VerticalMoment.`**`measures`**
Read-only tuplet of zero or more measures at vertical moment.

`VerticalMoment.`**`music`**
Read-only tuple of components in selection.

`VerticalMoment.`**`next_vertical_moment`**
Read-only reference to next vertical moment forward in time.

`VerticalMoment.`**`notes`**
Read-only tuple of zero or more notes at vertical moment.

`VerticalMoment.`**`offset`**
Read-only rational-valued score offset at which vertical moment is evaluated.

`VerticalMoment.`**`overlap_components`**
Read-only tuple of components in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.`**`overlap_leaves`**
Read-only tuple of leaves in vertical moment starting before vertical moment, ordered by score index.

`VerticalMoment.`**`overlap_measures`**
Read-only tuple of measures in vertical moment starting before vertical moment, ordered by score index.

VerticalMoment.**overlap_notes**
Read-only tuple of notes in vertical moment starting before vertical moment, ordered by score index.

VerticalMoment.**previous_vertical_moment**
Read-only reference to prev vertical moment backward in time.

VerticalMoment.**start_components**
Read-only tuple of components in vertical moment starting with at vertical moment, ordered by score index.

VerticalMoment.**start_leaves**
Read-only tuple of leaves in vertical moment starting with vertical moment, ordered by score index.

VerticalMoment.**start_notes**
Read-only tuple of notes in vertical moment starting with vertical moment, ordered by score index.

VerticalMoment.**storage_format**
Storage format of Abjad object.

Return string.

VerticalMoment.**timespan**
Read-only timespan of selection.

Special methods

VerticalMoment.**__add__**(*expr*)

VerticalMoment.**__contains__**(*expr*)

VerticalMoment.**__eq__**(*expr*)

VerticalMoment.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

VerticalMoment.**__getitem__**(*expr*)

VerticalMoment.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

VerticalMoment.**__hash__**()

VerticalMoment.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

VerticalMoment.**__len__**()

VerticalMoment.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

VerticalMoment.**__ne__**(*expr*)

VerticalMoment.**__radd__**(*expr*)

VerticalMoment.**__repr__**()

44.2 Functions

44.2.1 verticalitytools.get_vertical_moment_at_offset_in_expr

verticalitytools.**get_vertical_moment_at_offset_in_expr**(*expr*, *offset*)

New in version 2.0. Get vertical moment at *offset* in *expr*:

```
>>> score = Score([])
>>> staff = Staff(r"\times 4/3 { d''8 c''8 b'8 }")
>>> score.append(staff)
```

```
>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("a'4 g'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" f'8 e'8 d'8 c'8"""))
>>> score.append(piano_staff)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \fraction \times 4/3 {
            d''8
            c''8
            b'8
        }
    }
    \new PianoStaff <<
        \new Staff {
            a'4
            g'4
        }
        \new Staff {
            \clef "bass"
            f'8
            e'8
            d'8
            c'8
        }
    >>
>>
```

```
>>> args = (piano_staff, durationtools.Offset(1, 8))
```

```
>>> verticalitytools.get_vertical_moment_at_offset_in_expr(*args)
VerticalMoment(1/8, <<2>>)
```

```
>>> vertical_moment = _
>>> vertical_moment.leaves
(Note("a'4"), Note("e'8"))
```

Return vertical moment.

44.2.2 verticalitytools.get_vertical_moment_starting_with_component

verticalitytools.**get_vertical_moment_starting_with_component**(*component*, *governor=None*)

New in version 2.0. Get vertical moment starting with *component*:

```
>>> score = Score([])
>>> staff = Staff(r"\times 4/3 { d''8 c''8 b'8 }")
>>> score.append(staff)
```

```
>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("a'4 g'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" f'8 e'8 d'8 c'8"""))
>>> score.append(piano_staff)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \fraction \times 4/3 {
            d''8
            c''8
            b'8
        }
    }
    \new PianoStaff <<
        \new Staff {
            a'4
            g'4
        }
        \new Staff {
            \clef "bass"
            f'8
            e'8
            d'8
            c'8
        }
    >>
>>
```

```
>>> leaf = piano_staff[1][1]
```

```
>>> verticalitytools.get_vertical_moment_starting_with_component(leaf)
VerticalMoment(1/8, <<3>>)
```

Get vertical moment starting with *component* in *governor* when *governor* is not none:

```
>>> verticalitytools.get_vertical_moment_starting_with_component(leaf,
... governor=piano_staff)
VerticalMoment(1/8, <<2>>)
```

Return vertical moment.

44.2.3 verticalitytools.iterate_vertical_moments_in_expr

verticalitytools.**iterate_vertical_moments_in_expr**(*expr*, *reverse=False*)

New in version 2.10. Iterate vertical moments forward in *expr*:

```
>>> score = Score([])
>>> staff = Staff(r"\times 4/3 { d''8 c''8 b'8 }")
>>> score.append(staff)
```

```
>>> piano_staff = scoretools.PianoStaff([])
>>> piano_staff.append(Staff("a'4 g'4"))
>>> piano_staff.append(Staff(r"""\clef "bass" f'8 e'8 d'8 c'8"""))
>>> score.append(piano_staff)
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \fraction \times 4/3 {
            d''8
            c''8
            b'8
        }
    }
    \new PianoStaff <<
        \new Staff {
            a'4
            g'4
        }
        \new Staff {
            \clef "bass"
            f'8
            e'8
```

```
            d'8
            c'8
        }
    >>
>>
```

```
>>> for x in verticalitytools.iterate_vertical_moments_in_expr(score):
...     x.leaves
...
(Note("d''8"), Note("a'4"), Note("f'8"))
(Note("d''8"), Note("a'4"), Note("e'8"))
(Note("c''8"), Note("a'4"), Note("e'8"))
(Note("c''8"), Note("g'4"), Note("d'8"))
(Note("b'8"), Note("g'4"), Note("d'8"))
(Note("b'8"), Note("g'4"), Note("c'8"))
```

```
>>> for x in verticalitytools.iterate_vertical_moments_in_expr(piano_staff):
...     x.leaves
...
(Note("a'4"), Note("f'8"))
(Note("a'4"), Note("e'8"))
(Note("g'4"), Note("d'8"))
(Note("g'4"), Note("c'8"))
```

Iterate vertical moments backward in *expr*:

```
::
```

```
>>> for x in verticalitytools.iterate_vertical_moments_in_expr(score, reverse=True):
...     x.leaves
...
(Note("b'8"), Note("g'4"), Note("c'8"))
(Note("b'8"), Note("g'4"), Note("d'8"))
(Note("c''8"), Note("g'4"), Note("d'8"))
(Note("c''8"), Note("a'4"), Note("e'8"))
(Note("d''8"), Note("a'4"), Note("e'8"))
(Note("d''8"), Note("a'4"), Note("f'8"))
```

```
>>> for x in verticalitytools.iterate_vertical_moments_in_expr(piano_staff, reverse=True):
...     x.leaves
...
(Note("g'4"), Note("c'8"))
(Note("g'4"), Note("d'8"))
(Note("a'4"), Note("e'8"))
(Note("a'4"), Note("f'8"))
```

Return generator.

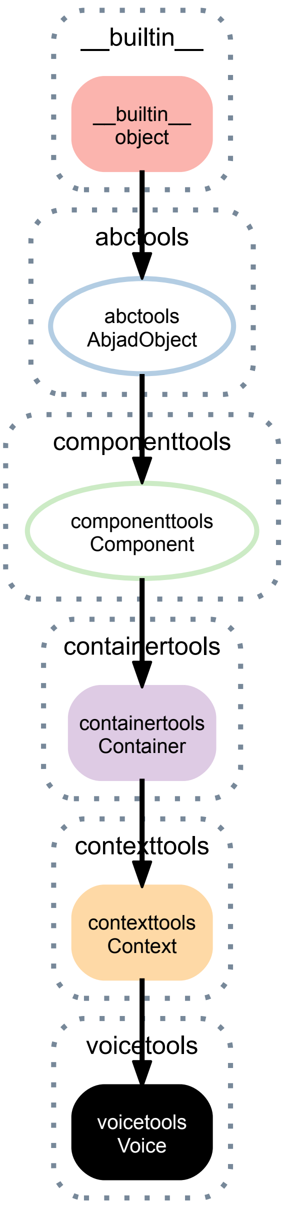
CHAPTER

FORTYFIVE

VOICETOOLS

45.1 Concrete Classes

45.1.1 voicetools.Voice



class voicetools.**Voice**(*music=None*, *context_name='Voice'*, *name=None*)

Abjad model of a voice:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
```

```
>>> voice
Voice{4}
```

```
>>> f(voice)
\new Voice {
    c'8
    d'8
    e'8
    f'8
}
```

```
>>> show(voice)
```

Return voice instance.

Read-only properties

Voice.**contents_duration**

Voice.**descendants**

Read-only reference to component descendants score selection.

Voice.**duration**

Voice.**duration_in_seconds**

Voice.**engraver_consists**

New in version 2.0. Unordered set of LilyPond engravers to include in context definition.

Manage with add, update, other standard set commands:

```
>>> staff = Staff([])
>>> staff.engraver_consists.append('Horizontal_bracket_engraver')
>>> f(staff)
\new Staff \with {
    \consists Horizontal_bracket_engraver
} {
}
```

Voice.**engraver_removals**

New in version 2.0. Unordered set of LilyPond engravers to remove from context.

Manage with add, update, other standard set commands.

```
>>> staff = Staff([])
>>> staff.engraver_removals.append('Time_signature_engraver')
>>> f(staff)
\new Staff \with {
    \remove Time_signature_engraver
} {
}
```

Voice.**is_semantic**

Voice.**leaves**

Read-only tuple of leaves in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.leaves
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple of zero or more leaves.

Voice.**lilypond_format**

Voice.**lineage**
Read-only reference to component lineage score selection.

Voice.**music**
Read-only tuple of components in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> container.music
(Note("c'8"), Note("d'8"), Note("e'8"))
```

Return tuple or zero or more components.

Voice.**override**
Read-only reference to LilyPond grob override component plug-in.

Voice.**parent**

Voice.**parentage**
Read-only reference to component parentage score selection.

Voice.**preprolated_duration**

Voice.**prolation**

Voice.**set**
Read-only reference LilyPond context setting component plug-in.

Voice.**spanners**
Read-only reference to unordered set of spanners attached to component.

Voice.**storage_format**
Storage format of Abjad object.

Return string.

Voice.**timespan**
Read-only timespan of component.

Voice.**timespan_in_seconds**
Read-only timespan of component in seconds.

Read/write properties

Voice.**context_name**
Read / write name of context as a string.

Voice.**is_nonsemantic**
Set indicator of nonsemantic voice:

```
>>> measures = measuretools.make_measures_with_full_measure_spacer_skips(
...     [(1, 8), (5, 16), (5, 16)])
>>> voice = Voice(measures)
>>> voice.name = 'HiddenTimeSignatureVoice'
```

```
>>> voice.is_nonsemantic = True
```

```
>>> f(voice)
\context Voice = "HiddenTimeSignatureVoice" {
    {
        \time 1/8
```

```
            s1 * 1/8
        }
        {
            \time 5/16
            s1 * 5/16
        }
        {
            s1 * 5/16
        }
}
```

```
>>> voice.is_nonsemantic
True
```

Get indicator of nonsemantic voice:

```
>>> voice = Voice([])
```

```
>>> voice.is_nonsemantic
False
```

Return boolean.

The intent of this read / write attribute is to allow composers to tag invisible voices used to house time signatures indications, bar number directives or other pieces of score-global non-musical information. Such nonsemantic voices can then be omitted from voice interation and other functions.

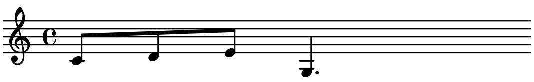
`Voice.`**`is_parallel`**

Get parallel container:

```
>>> container = Container([Voice("c'8 d'8 e'8"), Voice('g4.')])
```

```
>>> f(container)
{
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
}
```

```
>>> show(container)
```



```
>>> container.is_parallel
False
```

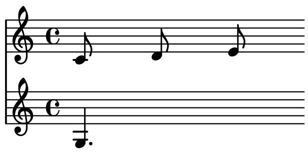
Return boolean.

Set parallel container:

```
>>> container.is_parallel = True
```

```
>>> f(container)
<<
    \new Voice {
        c'8
        d'8
        e'8
    }
    \new Voice {
        g4.
    }
>>
```

```
>>> show(container)
```



Return none.

`Voice.name`
Read-write name of context. Must be string or none.

Methods

`Voice.append(`*component*`)`
Append *component* to container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

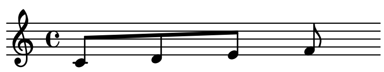
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.append(Note("f'8"))
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    f'8
}
```

```
>>> show(container)
```



Return none.

`Voice.extend(`*expr*`)`
Extend *expr* against container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

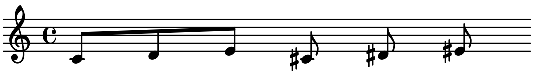
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.extend([Note("cs'8"), Note("ds'8"), Note("es'8")])
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
    cs'8
    ds'8
    es'8
}
```

```
>>> show(container)
```



Return none. New in version 2.3: `expr` may now be a LilyPond input string.

`Voice.`**`index`**(*component*)

Index *component* in container:

```
>>> container = Container("c'8 d'8 e'8")
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.index(note)
2
```

Return nonnegative integer.

`Voice.`**`insert`**(*i*, *component*)

Insert *component* in container at index *i*:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

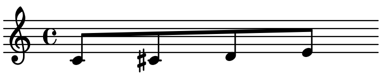
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.insert(1, Note("cs'8"))
```

```
>>> f(container)
{
    c'8 [
    cs'8
    d'8
    e'8 ]
}
```

```
>>> show(container)
```



Return none.

`Voice.pop(i=-1)`

Pop component at index *i* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

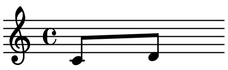
```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

```
>>> show(container)
```

```
>>> container.pop(-1)
Note("e'8")
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return component.

`Voice.remove(component)`

Remove *component* from container:

```
>>> container = Container("c'8 d'8 e'8")
>>> beam = beamtools.BeamSpanner(container.music)
```

```
>>> f(container)
{
    c'8 [
    d'8
    e'8 ]
}
```

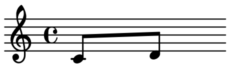
```
>>> show(container)
```

```
>>> note = container[-1]
>>> note
Note("e'8")
```

```
>>> container.remove(note)
```

```
>>> f(container)
{
    c'8 [
    d'8 ]
}
```

```
>>> show(container)
```



Return none.

Special methods

Voice.**__add__**(*expr*)
Concatenate containers self and expr. The operation c = a + b returns a new Container c with the content of both a and b. The operation is non-commutative: the content of the first operand will be placed before the content of the second operand.

Voice.**__contains__**(*expr*)
True if expr is in container, otherwise False.

Voice.**__copy__**(**args*)

Voice.**__delitem__**(*i*)
Find component(s) at index or slice 'i' in container. Detach component(s) from parentage. Withdraw component(s) from crossing spanners. Preserve spanners that component(s) cover(s).

Voice.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

Voice.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Voice.**__getitem__**(*i*)
Return component at index i in container. Shallow traversal of container for numeric indices only.

Voice.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Voice.**__iadd__**(*expr*)
__iadd__ avoids unnecessary copying of structures.

Voice.**__imul__**(*total*)
Multiply contents of container 'total' times. Return multiplied container.

Voice.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Voice.**__len__**()
Return nonnegative integer number of components in container.

Voice.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Voice.**__mul__**(*n*)

Voice.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Voice.**__radd__**(*expr*)
Extend container by contents of expr to the right.

Voice.**__repr__**()

Voice.**__rmul__**(*n*)

Voice.**__setitem__**(*i*, *expr*)
Set 'expr' in self at nonnegative integer index i. Or, set 'expr' in self at slice i. Find spanners that dominate self[i] and children of self[i]. Replace contents at self[i] with 'expr'. Reattach spanners to new contents. This operation always leaves score tree in tact.

45.2 Functions

45.2.1 voicetools.all_are_voices

voicetools.**all_are_voices**(*expr*)
New in version 2.6. True when *expr* is a sequence of Abjad voices:

```
>>> voice = Voice("c'4 d'4 e'4 f'4")
```

```
>>> voicetools.all_are_voices([voice])
True
```

True when *expr* is an empty sequence:

```
>>> voicetools.all_are_voices([])
True
```

Otherwise false:

```
>>> voicetools.all_are_voices('foo')
False
```

Return boolean.

Function wraps `componenttools.all_are_components()`.

45.2.2 voicetools.get_first_voice_in_improper_parentage_of_component

voicetools.**get_first_voice_in_improper_parentage_of_component**(*component*)
New in version 2.0. Get first voice in improper parentage of *component*:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> staff = Staff([voice])
```

```
>>> f(staff)
\new Staff {
    \new Voice {
        c'8
        d'8
        e'8
        f'8
    }
}
```

```
>>> voicetools.get_first_voice_in_improper_parentage_of_component(staff.leaves[0])
Voice{4}
```

Return voice or none.

45.2.3 voicetools.get_first_voice_in_proper_parentage_of_component

voicetools.**get_first_voice_in_proper_parentage_of_component**(*component*)

New in version 2.0. Get first voice in proper parentage of *component*:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> staff = Staff([voice])
```

```
>>> f(staff)
\new Staff {
    \new Voice {
        c'8
        d'8
        e'8
        f'8
    }
}
```

```
>>> voicetools.get_first_voice_in_proper_parentage_of_component(staff.leaves[0])
Voice{4}
```

Return voice or none.

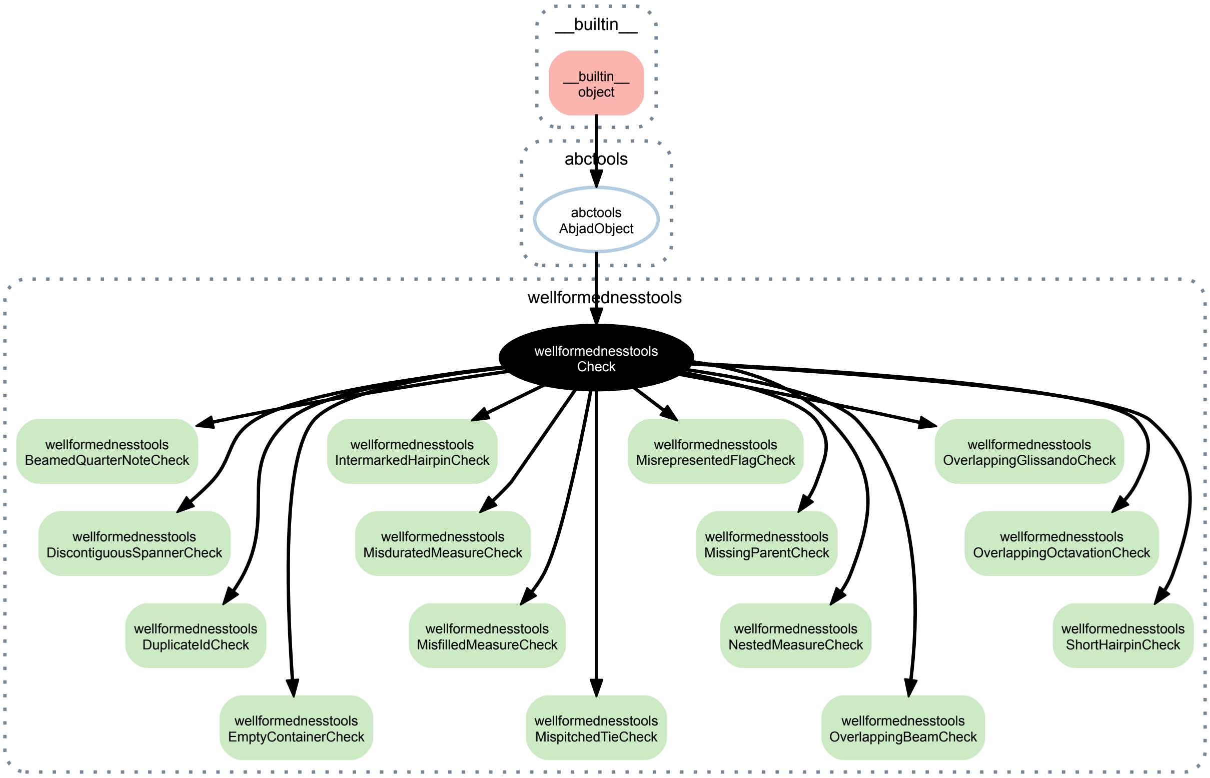
CHAPTER

FORTYSIX

WELLFORMEDNESSTOOLS

46.1 Abstract Classes

46.1.1 wellformednesstools.Check



class `wellformednesstools.`**`Check`**

Read-only properties

`Check.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`Check.`**`check`**(*expr*)

`Check.`**`report`**(*expr*)

`Check.`**`violators`**`(`*expr*`)`

Special methods

`Check.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Check.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Check.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Check.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Check.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Check.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

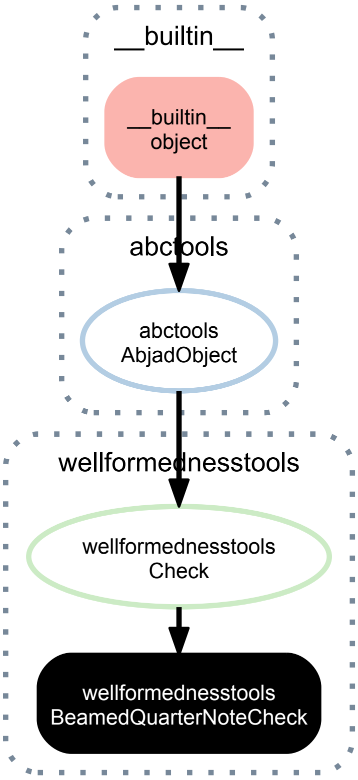
Return boolean.

`Check.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

46.2 Concrete Classes

46.2.1 wellformednesstools.BeamedQuarterNoteCheck



class `wellformednesstools.`**`BeamedQuarterNoteCheck`**

Read-only properties

`BeamedQuarterNoteCheck.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`BeamedQuarterNoteCheck.`**`check`**(*expr*)

`BeamedQuarterNoteCheck.`**`report`**(*expr*)

`BeamedQuarterNoteCheck.`**`violators`**(*expr*)

Special methods

`BeamedQuarterNoteCheck.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`BeamedQuarterNoteCheck.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`BeamedQuarterNoteCheck.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`BeamedQuarterNoteCheck.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`BeamedQuarterNoteCheck.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

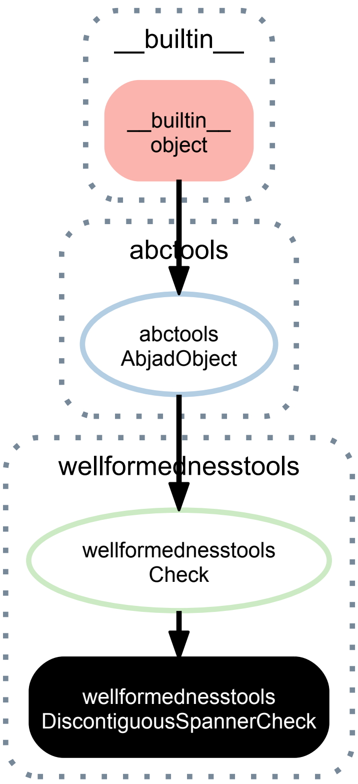
`BeamedQuarterNoteCheck.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`BeamedQuarterNoteCheck.__repr__()`
Interpreter representation of Abjad object.

Return string.

46.2.2 wellformednesstools.DiscontiguousSpannerCheck



class `wellformednesstools.DiscontiguousSpannerCheck`
There are now two different types of spanner. Most spanners demand that spanner components be thread-contiguous. But a few special spanners (like Tempo) do not make such a demand. The check here consults the experimental *_contiguity_constraint*.

Read-only properties

`DiscontiguousSpannerCheck.storage_format`
Storage format of Abjad object.

Return string.

Methods

`DiscontiguousSpannerCheck.`**`check`**`(`*expr*`)`

`DiscontiguousSpannerCheck.`**`report`**`(`*expr*`)`

`DiscontiguousSpannerCheck.`**`violators`**`(`*expr*`)`

Special methods

`DiscontiguousSpannerCheck.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`DiscontiguousSpannerCheck.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DiscontiguousSpannerCheck.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`DiscontiguousSpannerCheck.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`DiscontiguousSpannerCheck.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

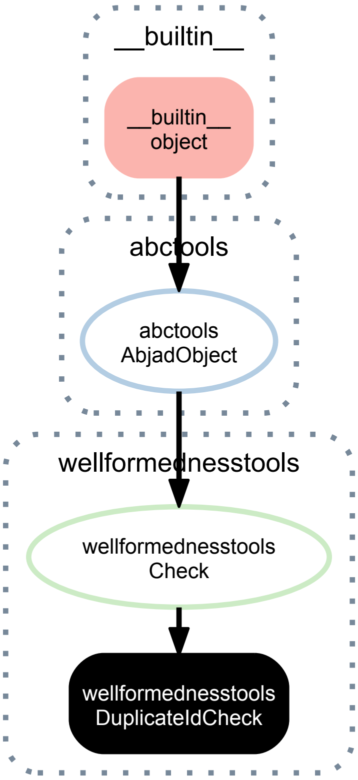
`DiscontiguousSpannerCheck.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`DiscontiguousSpannerCheck.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

46.2.3 wellformednesstools.DuplicateIdCheck



class `wellformednesstools.`**`DuplicateIdCheck`**

Read-only properties

`DuplicateIdCheck.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`DuplicateIdCheck.`**`check`**(*expr*)

`DuplicateIdCheck.`**`report`**(*expr*)

`DuplicateIdCheck.`**`violators`**(*expr*)

Special methods

`DuplicateIdCheck.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`DuplicateIdCheck.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`DuplicateIdCheck.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

DuplicateIdCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DuplicateIdCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

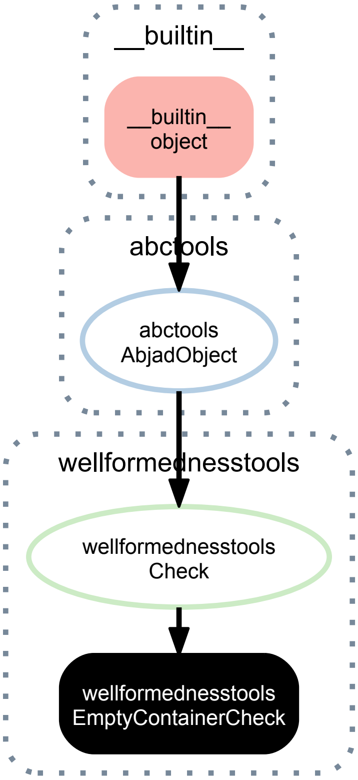
DuplicateIdCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DuplicateIdCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.4 wellformednesstools.EmptyContainerCheck



class wellformednesstools.**EmptyContainerCheck**

Read-only properties

EmptyContainerCheck.**storage_format**
Storage format of Abjad object.

Return string.

Methods

EmptyContainerCheck.**check**(*expr*)

EmptyContainerCheck.**report**(*expr*)

EmptyContainerCheck.**violators**(*expr*)

Special methods

EmptyContainerCheck.**__eq__**(*expr*)
True when id(self) equals id(expr).

Return boolean.

EmptyContainerCheck.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

EmptyContainerCheck.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

EmptyContainerCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

EmptyContainerCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

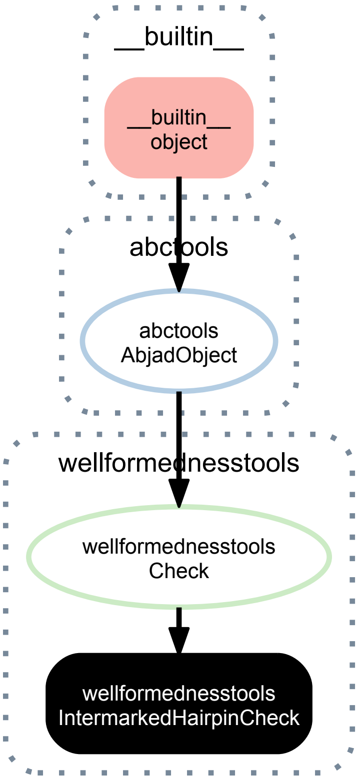
EmptyContainerCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

EmptyContainerCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.5 wellformednesstools.IntermarkedHairpinCheck



class `wellformednesstools.`**`IntermarkedHairpinCheck`**
Are there any dynamic marks in the middle of a hairpin?

Read-only properties

`IntermarkedHairpinCheck.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`IntermarkedHairpinCheck.`**`check`**(*expr*)

`IntermarkedHairpinCheck.`**`report`**(*expr*)

`IntermarkedHairpinCheck.`**`violators`**(*expr*)

Special methods

`IntermarkedHairpinCheck.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`IntermarkedHairpinCheck.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

IntermarkedHairpinCheck.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

IntermarkedHairpinCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

IntermarkedHairpinCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

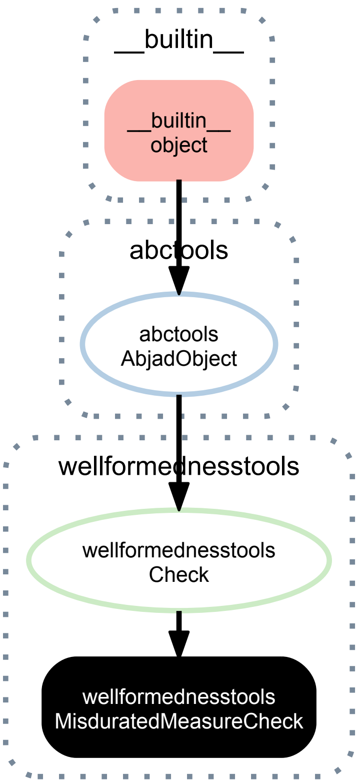
IntermarkedHairpinCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

IntermarkedHairpinCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.6 wellformednesstools.MisduratedMeasureCheck



class wellformednesstools.**MisduratedMeasureCheck**
Does the (pre)prolated duration of the measure match its time signature?

Read-only properties

MisduratedMeasureCheck.**storage_format**
Storage format of Abjad object.

Return string.

Methods

`MisduratedMeasureCheck.`**`check`**(*expr*)

`MisduratedMeasureCheck.`**`report`**(*expr*)

`MisduratedMeasureCheck.`**`violators`**(*expr*)

Special methods

`MisduratedMeasureCheck.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MisduratedMeasureCheck.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MisduratedMeasureCheck.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`MisduratedMeasureCheck.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MisduratedMeasureCheck.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

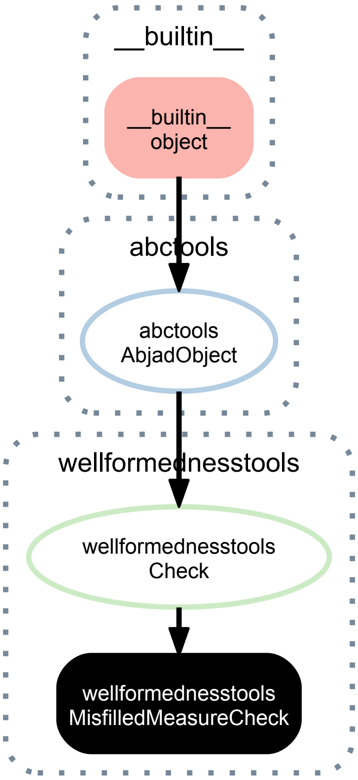
`MisduratedMeasureCheck.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`MisduratedMeasureCheck.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

46.2.7 wellformednesstools.MisfilledMeasureCheck



class `wellformednesstools.`**`MisfilledMeasureCheck`**
Check that time signature duration equals measure contents duration for every measure.

Read-only properties

`MisfilledMeasureCheck.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`MisfilledMeasureCheck.`**`check`**(*expr*)

`MisfilledMeasureCheck.`**`report`**(*expr*)

`MisfilledMeasureCheck.`**`violators`**(*expr*)

Special methods

`MisfilledMeasureCheck.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MisfilledMeasureCheck.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MisfilledMeasureCheck.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MisfilledMeasureCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MisfilledMeasureCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

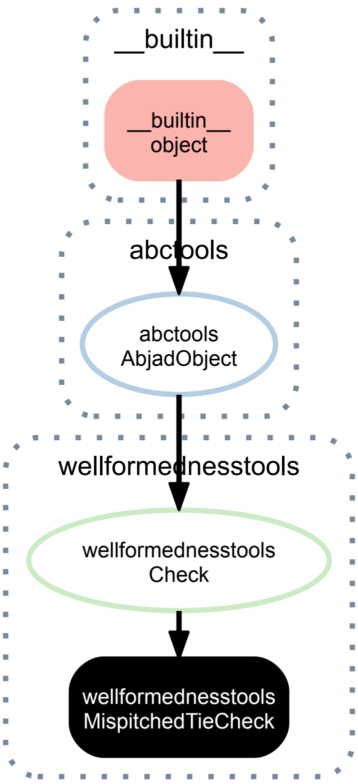
MisfilledMeasureCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MisfilledMeasureCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.8 wellformednesstools.MispitchedTieCheck



class wellformednesstools.**MispitchedTieCheck**

Read-only properties

MispitchedTieCheck.**storage_format**
Storage format of Abjad object.

Return string.

Methods

`MispitchedTieCheck.`**`check`**`(`*expr*`)`

`MispitchedTieCheck.`**`report`**`(`*expr*`)`

`MispitchedTieCheck.`**`violators`**`(`*expr*`)`

Special methods

`MispitchedTieCheck.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`MispitchedTieCheck.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MispitchedTieCheck.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`MispitchedTieCheck.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MispitchedTieCheck.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

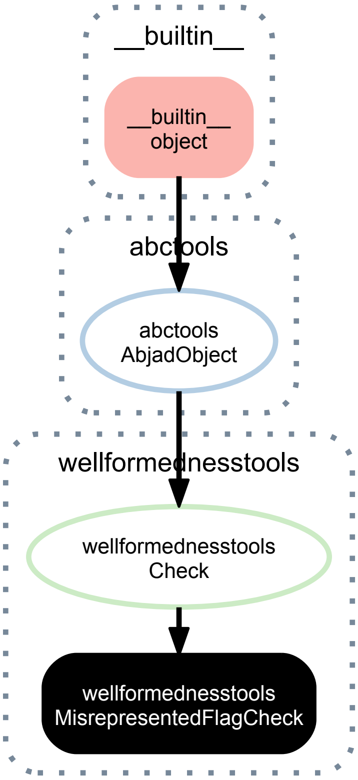
`MispitchedTieCheck.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`MispitchedTieCheck.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

46.2.9 wellformednesstools.MisrepresentedFlagCheck



class `wellformednesstools.`**`MisrepresentedFlagCheck`**

Read-only properties

`MisrepresentedFlagCheck.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`MisrepresentedFlagCheck.`**`check`**(*expr*)

`MisrepresentedFlagCheck.`**`report`**(*expr*)

`MisrepresentedFlagCheck.`**`violators`**(*expr*)

Special methods

`MisrepresentedFlagCheck.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MisrepresentedFlagCheck.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MisrepresentedFlagCheck.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MisrepresentedFlagCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MisrepresentedFlagCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

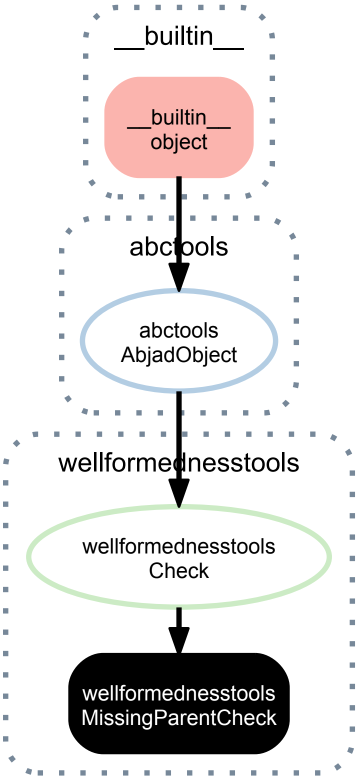
MisrepresentedFlagCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MisrepresentedFlagCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.10 wellformednesstools.MissingParentCheck



class wellformednesstools.**MissingParentCheck**
Each node except the root needs a parent.

Read-only properties

MissingParentCheck.**storage_format**
Storage format of Abjad object.

Return string.

Methods

MissingParentCheck.**check**(*expr*)

`MissingParentCheck.`**`report`**`(`*expr*`)`

`MissingParentCheck.`**`violators`**`(`*expr*`)`

Special methods

`MissingParentCheck.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`MissingParentCheck.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MissingParentCheck.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`MissingParentCheck.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`MissingParentCheck.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

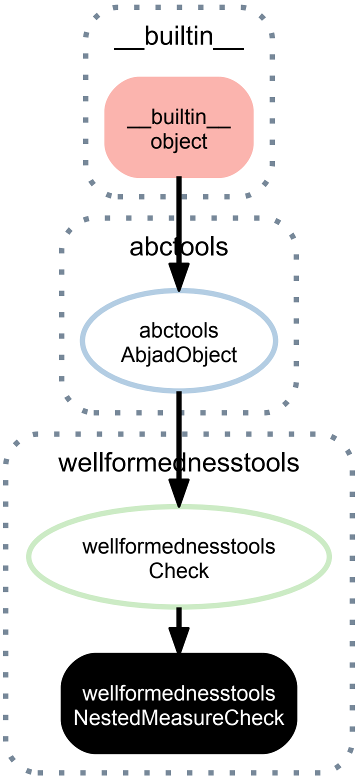
`MissingParentCheck.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`MissingParentCheck.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

46.2.11 wellformednesstools.NestedMeasureCheck



class `wellformednesstools.`**`NestedMeasureCheck`**

Do we have any nested measures?

Read-only properties

`NestedMeasureCheck.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`NestedMeasureCheck.`**`check`**(*expr*)

`NestedMeasureCheck.`**`report`**(*expr*)

`NestedMeasureCheck.`**`violators`**(*expr*)

Special methods

`NestedMeasureCheck.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`NestedMeasureCheck.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

NestedMeasureCheck.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

NestedMeasureCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

NestedMeasureCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

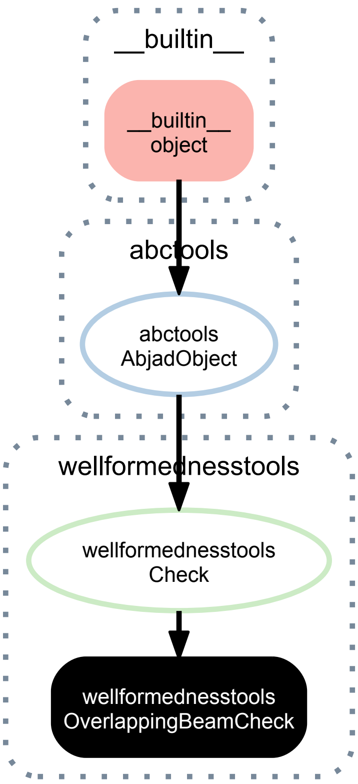
NestedMeasureCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

NestedMeasureCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.12 wellformednesstools.OverlappingBeamCheck



class wellformednesstools.**OverlappingBeamCheck**
Beams must not overlap.

Read-only properties

OverlappingBeamCheck.**storage_format**
Storage format of Abjad object.

Return string.

Methods

OverlappingBeamCheck.**check**(*expr*)

OverlappingBeamCheck.**report**(*expr*)

OverlappingBeamCheck.**violators**(*expr*)

Special methods

OverlappingBeamCheck.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

OverlappingBeamCheck.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OverlappingBeamCheck.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

OverlappingBeamCheck.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OverlappingBeamCheck.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

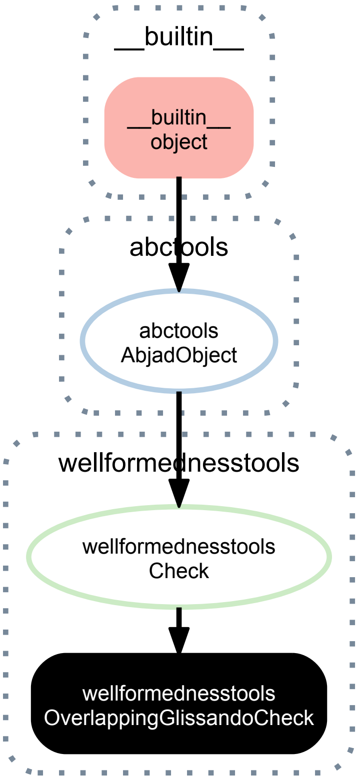
OverlappingBeamCheck.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

OverlappingBeamCheck.**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.13 wellformednesstools.OverlappingGlissandoCheck



class `wellformednesstools.`**`OverlappingGlissandoCheck`**

Glissandi must not overlap. Dove-tailed glissandi are OK.

Read-only properties

`OverlappingGlissandoCheck.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`OverlappingGlissandoCheck.`**`check`**(*expr*)

`OverlappingGlissandoCheck.`**`report`**(*expr*)

`OverlappingGlissandoCheck.`**`violators`**(*expr*)

Special methods

`OverlappingGlissandoCheck.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`OverlappingGlissandoCheck.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`OverlappingGlissandoCheck.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`OverlappingGlissandoCheck.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`OverlappingGlissandoCheck.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

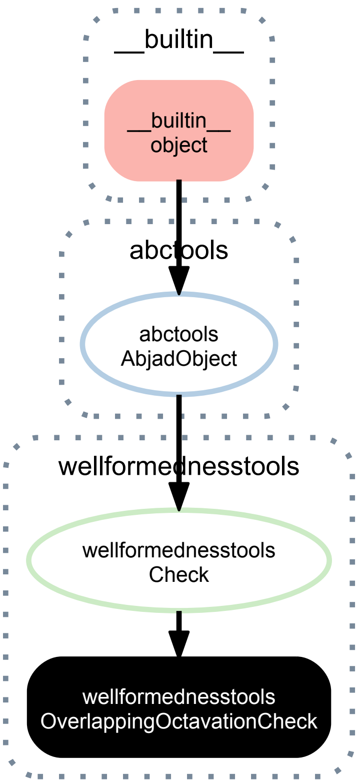
`OverlappingGlissandoCheck.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`OverlappingGlissandoCheck.__repr__()`
Interpreter representation of Abjad object.

Return string.

46.2.14 wellformednesstools.OverlappingOctavationCheck



class `wellformednesstools.OverlappingOctavationCheck`
Octavation spanners must not overlap.

Read-only properties

`OverlappingOctavationCheck.storage_format`
Storage format of Abjad object.

Return string.

Methods

`OverlappingOctavationCheck.`**check**(*expr*)

`OverlappingOctavationCheck.`**report**(*expr*)

`OverlappingOctavationCheck.`**violators**(*expr*)

Special methods

`OverlappingOctavationCheck.`**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`OverlappingOctavationCheck.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OverlappingOctavationCheck.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`OverlappingOctavationCheck.`**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`OverlappingOctavationCheck.`**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

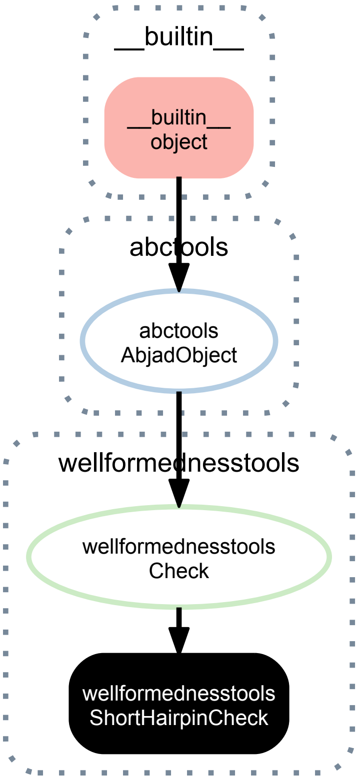
`OverlappingOctavationCheck.`**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`OverlappingOctavationCheck.`**__repr__**()
Interpreter representation of Abjad object.

Return string.

46.2.15 wellformednesstools.ShortHairpinCheck



class `wellformednesstools.`**`ShortHairpinCheck`**

Hairpins must span at least two leaves.

Read-only properties

`ShortHairpinCheck.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`ShortHairpinCheck.`**`check`**(*expr*)

`ShortHairpinCheck.`**`report`**(*expr*)

`ShortHairpinCheck.`**`violators`**(*expr*)

Special methods

`ShortHairpinCheck.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`ShortHairpinCheck.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`ShortHairpinCheck.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`ShortHairpinCheck.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ShortHairpinCheck.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ShortHairpinCheck.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`ShortHairpinCheck.__repr__()`
Interpreter representation of Abjad object.

Return string.

46.3 Functions

46.3.1 wellformednesstools.is_well_formed_component

`wellformednesstools.is_well_formed_component(expr, allow_empty_containers=True)`
New in version 1.1. True when *component* is well formed:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'8, e'8, f'8)
>>> wellformednesstools.is_well_formed_component(staff)
True
```

Otherwise false:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'4, e'8, f'8)
>>> wellformednesstools.is_well_formed_component(staff)
False
```

Beamed quarter notes are not well formed.

Return boolean.

46.3.2 wellformednesstools.list_badly_formed_components_in_expr

`wellformednesstools.list_badly_formed_components_in_expr(expr)`
New in version 1.1. List badly formed components in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'4, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'4
    e'8
    f'8 ]
```

```
}
>>> wellformednesstools.list_badly_formed_components_in_expr(staff)
[Note("d'4")]
```

Beamed quarter notes are not well formed.

Return newly created list of zero or more components.

46.3.3 wellformednesstools.list_checks

wellformednesstools.**list_checks**()

New in version 2.8. List checks:

```
>>> for check in wellformednesstools.list_checks(): check
...
BeamedQuarterNoteCheck()
DiscontiguousSpannerCheck()
DuplicateIdCheck()
EmptyContainerCheck()
IntermarkedHairpinCheck()
MisduratedMeasureCheck()
MisfilledMeasureCheck()
MispitchedTieCheck()
MisrepresentedFlagCheck()
MissingParentCheck()
NestedMeasureCheck()
OverlappingBeamCheck()
OverlappingGlissandoCheck()
OverlappingOctavationCheck()
ShortHairpinCheck()
```

Return list of checks.

46.3.4 wellformednesstools.tabulate_well_formedness_violations_in_expr

wellformednesstools.**tabulate_well_formedness_violations_in_expr**(*expr*)

New in version 1.1. Tabulate well-formedness violations in *expr*:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> staff[1].written_duration = Duration(1, 4)
>>> beamtools.BeamSpanner(staff[:])
BeamSpanner(c'8, d'4, e'8, f'8)
>>> f(staff)
\new Staff {
    c'8 [
    d'4
    e'8
    f'8 ]
}
```

```
>>> wellformednesstools.tabulate_well_formedness_violations_in_expr(staff)
   1 /    4 beamed quarter note
   0 /    1 discontiguous spanner
   0 /    5 duplicate id
   0 /    1 empty container
   0 /    0 intermarked hairpin
   0 /    0 misdurated measure
   0 /    0 misfilled measure
   0 /    4 mispitched tie
   0 /    4 misrepresented flag
   0 /    5 missing parent
   0 /    0 nested measure
   0 /    0 overlapping beam
   0 /    0 overlapping glissando
   0 /    0 overlapping octavation
   0 /    0 short hairpin
```

Beamed quarter notes are not well formed.

Part II

Demos and example packages

CHAPTER

FORTYSEVEN

DESORDRE

47.1 Functions

47.1.1 desordre.make_desordre_cell

desordre.**make_desordre_cell**(*pitches*)

The function constructs and returns a *Désordre cell*. *pitches* is a list of numbers or, more generally, pitch tokens.

47.1.2 desordre.make_desordre_lilypond_file

desordre.**make_desordre_lilypond_file**()

47.1.3 desordre.make_desordre_measure

desordre.**make_desordre_measure**(*pitches*)

Constructs a measure composed of *Désordre cells*.

pitches is a list of lists of number (e.g., [[1, 2, 3], [2, 3, 4]])

The function returns a measure.

47.1.4 desordre.make_desordre_pitches

desordre.**make_desordre_pitches**()

47.1.5 desordre.make_desordre_score

desordre.**make_desordre_score**(*pitches*)

Returns a complete PianoStaff with Ligeti music!

47.1.6 desordre.make_desordre_staff

desordre.**make_desordre_staff**(*pitches*)

CHAPTER

FORTYEIGHT

FERNEYHOUGH

48.1 Functions

48.1.1 ferneyhough.configure_lilypond_file

ferneyhough.**configure_lilypond_file**(*lilypond_file*)

48.1.2 ferneyhough.configure_score

ferneyhough.**configure_score**(*score*)

48.1.3 ferneyhough.make_lilypond_file

ferneyhough.**make_lilypond_file**(*tuplet_duration*, *row_count*, *column_count*)

48.1.4 ferneyhough.make_nested_tuplet

ferneyhough.**make_nested_tuplet**(*tuplet_duration*, *outer_tuplet_proportions*, *inner_tuplet_subdivision_count*)

48.1.5 ferneyhough.make_row_of_nested_tuplets

ferneyhough.**make_row_of_nested_tuplets**(*tuplet_duration*, *outer_tuplet_proportions*, *column_count*)

48.1.6 ferneyhough.make_rows_of_nested_tuplets

ferneyhough.**make_rows_of_nested_tuplets**(*tuplet_duration*, *row_count*, *column_count*)

CHAPTER

FORTYNINE

MOZART

49.1 Functions

49.1.1 mozart.choose_mozart_measures

mozart.**choose_mozart_measures**()

49.1.2 mozart.make_mozart_lilypond_file

mozart.**make_mozart_lilypond_file**()

49.1.3 mozart.make_mozart_measure

mozart.**make_mozart_measure**(*measure_dict*)

49.1.4 mozart.make_mozart_measure_corpus

mozart.**make_mozart_measure_corpus**()

49.1.5 mozart.make_mozart_score

mozart.**make_mozart_score**()

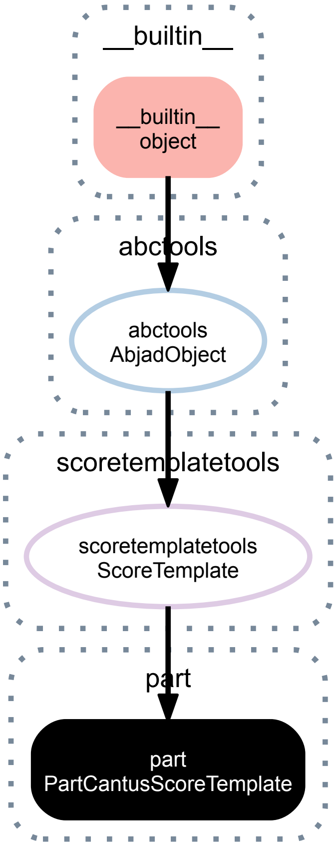
CHAPTER

FIFTY

PART

50.1 Concrete Classes

50.1.1 part.PartCantusScoreTemplate



class `part.`**`PartCantusScoreTemplate`**

Read-only properties

`PartCantusScoreTemplate.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`PartCantusScoreTemplate.__call__()`

`PartCantusScoreTemplate.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`PartCantusScoreTemplate.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PartCantusScoreTemplate.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`PartCantusScoreTemplate.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PartCantusScoreTemplate.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`PartCantusScoreTemplate.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`PartCantusScoreTemplate.__repr__()`
Interpreter representation of Abjad object.

Return string.

50.2 Functions

50.2.1 part.add_bell_music_to_score

`part.add_bell_music_to_score(score)`

50.2.2 part.add_string_music_to_score

`part.add_string_music_to_score(score)`

50.2.3 part.apply_bowing_marks

`part.apply_bowing_marks(score)`

50.2.4 part.apply_dynamic_marks

`part.apply_dynamic_marks(score)`

50.2.5 part.apply_expressive_marks

`part.apply_expressive_marks(score)`

50.2.6 part.apply_final_bar_lines

part.**apply_final_bar_lines**(*score*)

50.2.7 part.apply_page_breaks

part.**apply_page_breaks**(*score*)

50.2.8 part.apply_rehearsal_marks

part.**apply_rehearsal_marks**(*score*)

50.2.9 part.configure_lilypond_file

part.**configure_lilypond_file**(*lilypond_file*)

50.2.10 part.configure_score

part.**configure_score**(*score*)

50.2.11 part.create_pitch_contour_reservoir

part.**create_pitch_contour_reservoir**()

50.2.12 part.durate_pitch_contour_reservoir

part.**durate_pitch_contour_reservoir**(*pitch_contour_reservoir*)

50.2.13 part.edit_bass_voice

part.**edit_bass_voice**(*score*, *durated_reservoir*)

50.2.14 part.edit_cello_voice

part.**edit_cello_voice**(*score*, *durated_reservoir*)

50.2.15 part.edit_first_violin_voice

part.**edit_first_violin_voice**(*score*, *durated_reservoir*)

50.2.16 part.edit_second_violin_voice

part.**edit_second_violin_voice**(*score*, *durated_reservoir*)

50.2.17 part.edit_viola_voice

part.**edit_viola_voice**(*score*, *durated_reservoir*)

50.2.18 part.make_part_lilypond_file

part.**make_part_lilypond_file**()

50.2.19 part.shadow_pitch_contour_reservoir

part.**shadow_pitch_contour_reservoir**(*pitch_contour_reservoir*)

Part III

Abjad internal packages

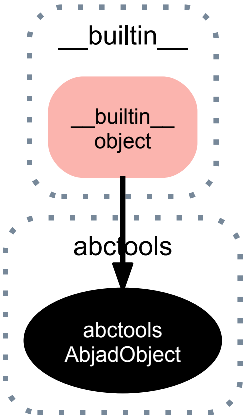
CHAPTER

FIFTYONE

ABCTOOLS

51.1 Abstract Classes

51.1.1 abctools.AbjadObject



class abctools.**AbjadObject**(**args*, ***kwargs*)

New in version 2.0. Abstract base class from which all custom classes should inherit.

Abajd objects raise exceptions on `__gt__`, `__ge__`, `__lt__`, `__le__`.

Abjad objects compare equal only with equal object IDs.

Authors of custom classes should override these behaviors as required.

Read-only properties

AbjadObject.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

AbjadObject.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

AbjadObject.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

AbjadObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjadObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

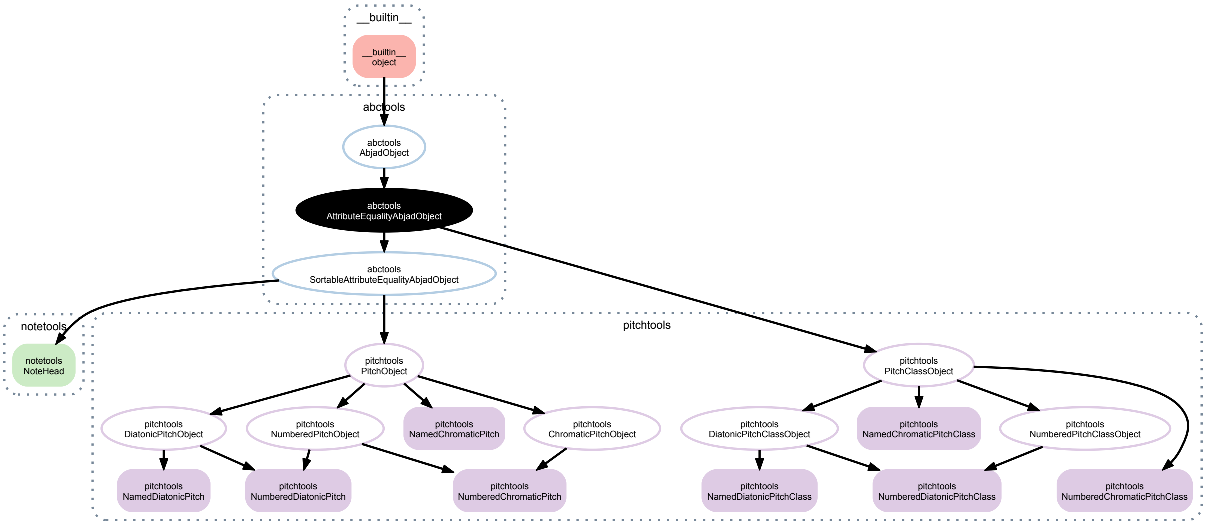
AbjadObject.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjadObject.**__repr__**()
Interpreter representation of Abjad object.

Return string.

51.1.2 abctools.AttributeEqualityAbjadObject



class abctools.**AttributeEqualityAbjadObject**(**args*, ***kwargs*)
New in version 2.0. Abstract base class to confer nonsorting attribute-equality to any custom class.

Nonsorting objects raise exceptions on `__gt__`, `__ge__`, `__lt__`, `__le__`.

Attribute-equality objects compare equal only with equal comparison attributes.

Read-only properties

AttributeEqualityAbjadObject.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

AttributeEqualityAbjadObject.**__eq__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

AttributeEqualityAbjadObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AttributeEqualityAbjadObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AttributeEqualityAbjadObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AttributeEqualityAbjadObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

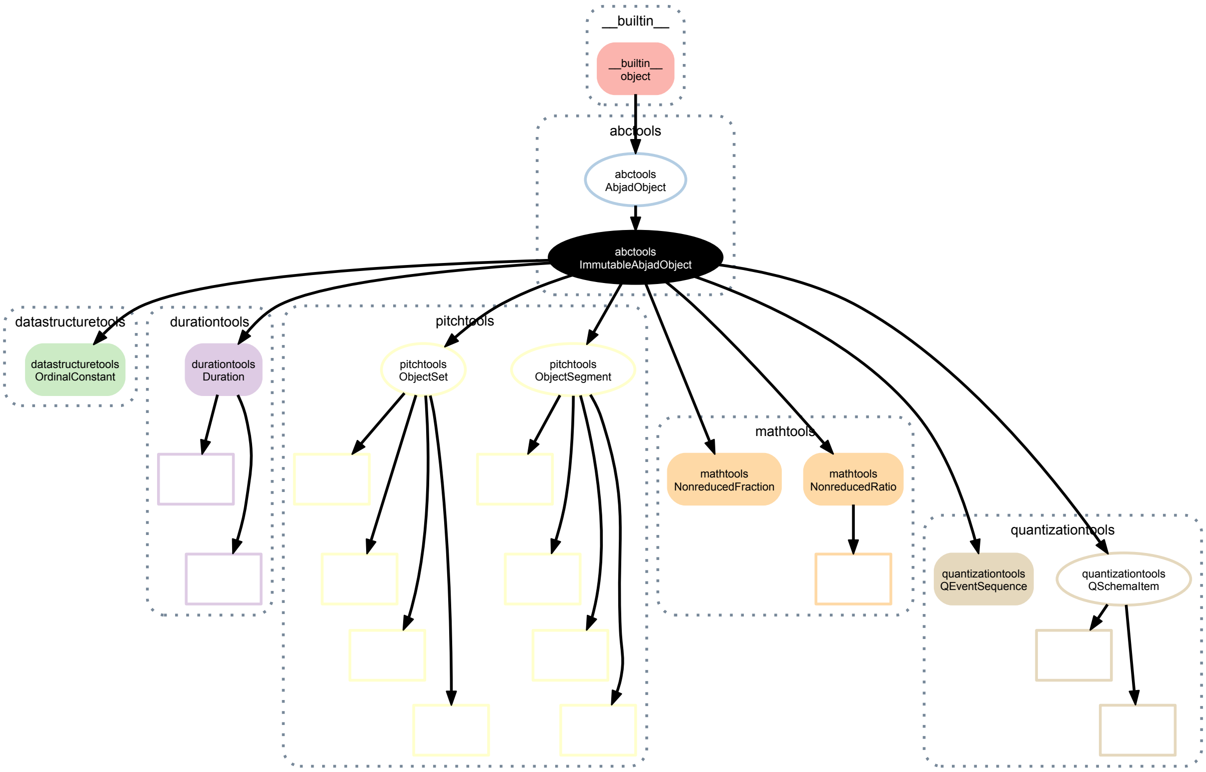
AttributeEqualityAbjadObject.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

AttributeEqualityAbjadObject.**__repr__**()
Interpreter representation of object defined equal to class name and format string.

Return string.

51.1.3 abctools.ImmutableAbjadObject



class abctools.**ImmutableAbjadObject**(**args*, ***kwargs*)
New in version 2.8. Abstract base class from which all custom classes which also subclass immutable builtin classes, such as tuple and frozenset, should inherit.

Read-only properties

`ImmutableAbjadObject.`**storage_format**
Storage format of Abjad object.

Return string.

Special methods

`ImmutableAbjadObject.`**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`ImmutableAbjadObject.`**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ImmutableAbjadObject.`**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`ImmutableAbjadObject.`**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ImmutableAbjadObject.`**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

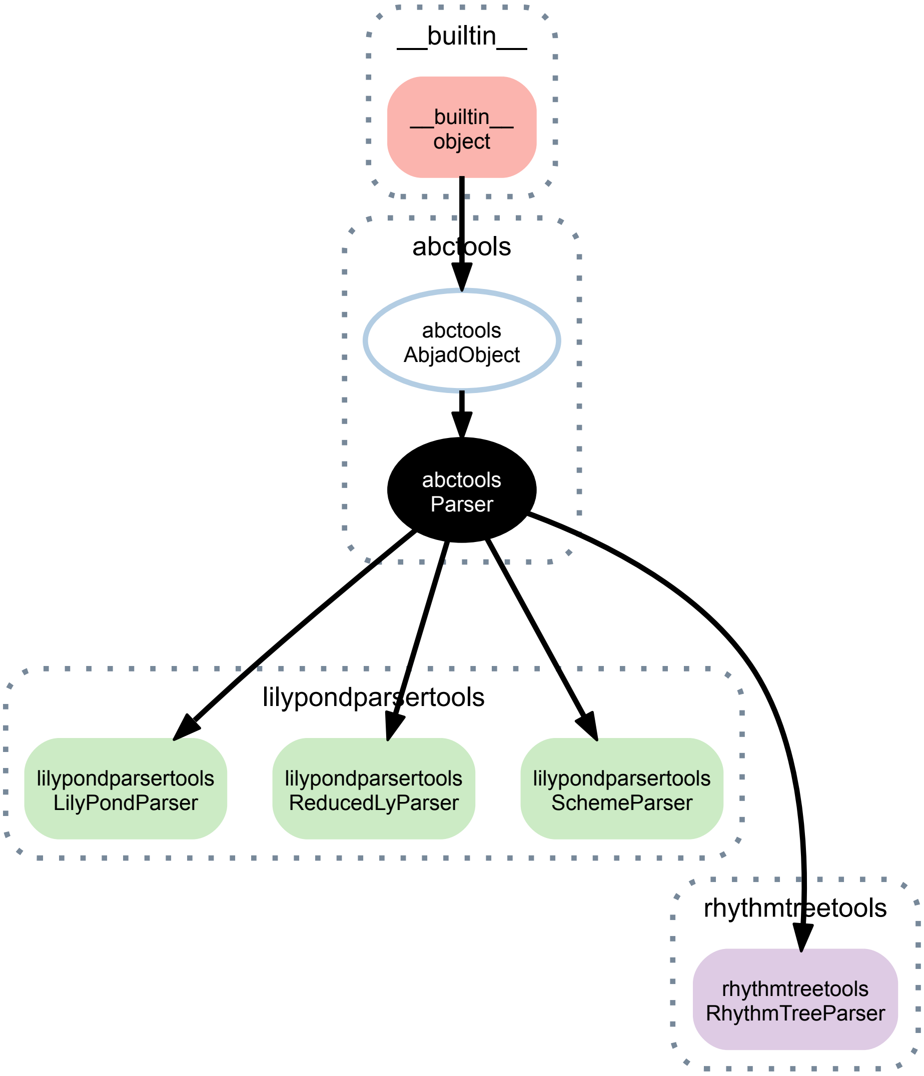
`ImmutableAbjadObject.`**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`ImmutableAbjadObject.`**__repr__**()
Interpreter representation of Abjad object.

Return string.

51.1.4 abctools.Parser



class `abctools.`**`Parser`**`(`*debug=False*`)`

Abstract base class for Abjad parsers.

Rules objects for lexing and parsing must be defined by overriding the abstract properties *lexer_rules_object* and *parser_rules_object*.

For most parsers these properties should simply return *self*.

Read-only properties

`Parser.`**`debug`**

True if the parser runs in debugging mode.

`Parser.`**`lexer`**

The parser's PLY Lexer instance.

`Parser.`**`lexer_rules_object`**

The object containing the parser's lexical rule definitions.

`Parser.`**`logger`**

The parser's Logger instance.

`Parser.`**`logger_path`**

The output path for the parser's logfile.

`Parser.`**`output_path`**
The output path for files associated with the parser.

`Parser.`**`parser`**
The parser's PLY LRParser instance.

`Parser.`**`parser_rules_object`**
The object containing the parser's syntactical rule definitions.

`Parser.`**`pickle_path`**
The output path for the parser's pickled parsing tables.

`Parser.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`Parser.`**`tokenize`**`(`*input_string*`)`
Tokenize *input string* and print results.

Special methods

`Parser.`**`__call__`**`(`*input_string*`)`
Parse *input_string* and return result.

`Parser.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Parser.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Parser.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Parser.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Parser.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

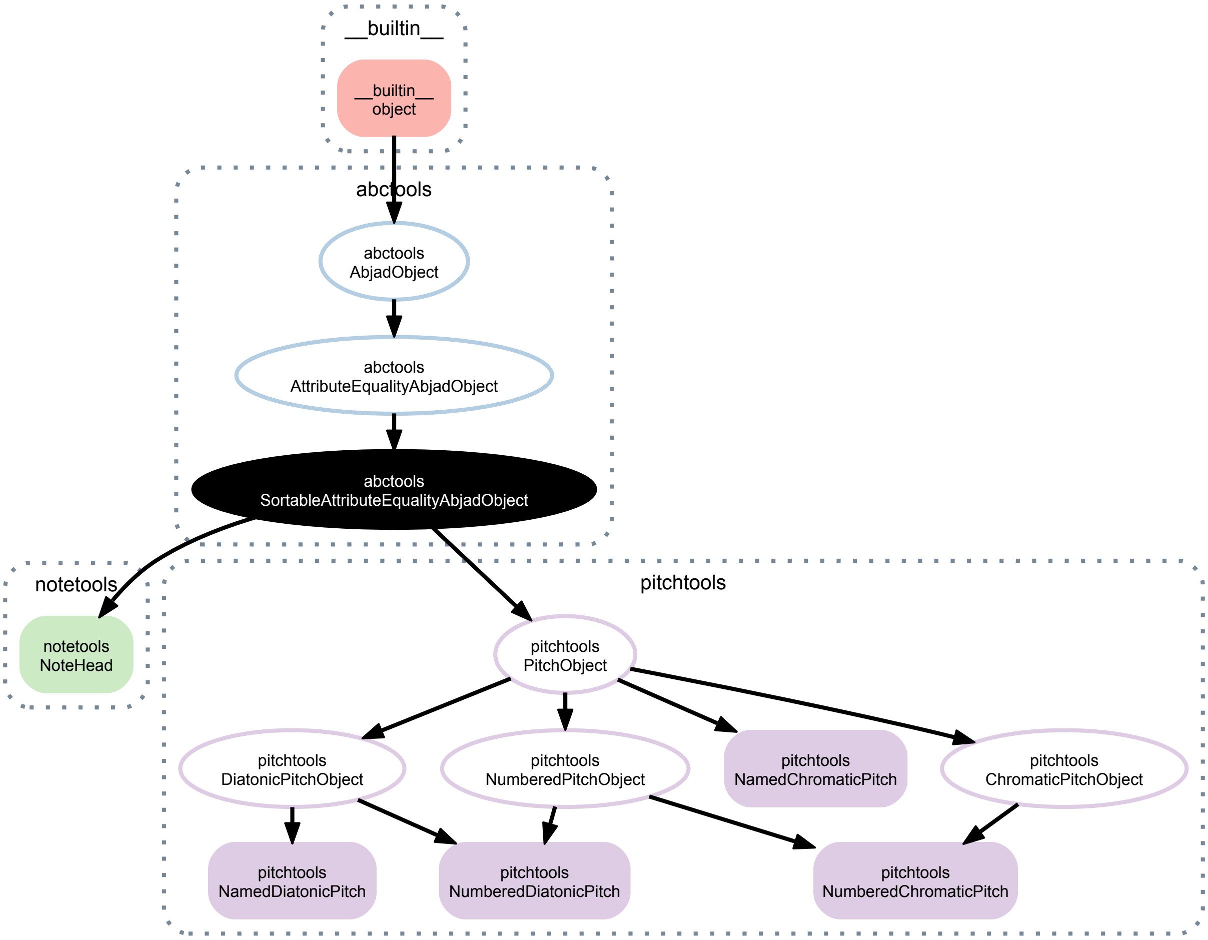
`Parser.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`Parser.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

51.1.5 abctools.SortableAttributeEqualityAbjadObject



class `abctools.`**`SortableAttributeEqualityAbjadObject`**`(`**args*, ***kwargs*`)`

New in version 2.0. Abstact base class to confer sortable attribute-equality to any custom class.

Sortable attribute-equality comparators implement `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__lt__`, `__le__` against a single comparison attribute.

Read-only properties

`SortableAttributeEqualityAbjadObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`SortableAttributeEqualityAbjadObject.`**`__eq__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`SortableAttributeEqualityAbjadObject.`**`__ge__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

`SortableAttributeEqualityAbjadObject.`**`__gt__`**`(`*arg*`)`

Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

SortableAttributeEqualityAbjadObject.**__le__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

SortableAttributeEqualityAbjadObject.**__lt__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

SortableAttributeEqualityAbjadObject.**__ne__**(*arg*)
Initialize new object from *arg* and evaluate comparison attributes.

Return boolean.

SortableAttributeEqualityAbjadObject.**__repr__**()
Interpreter representation of object defined equal to class name and format string.

Return string.

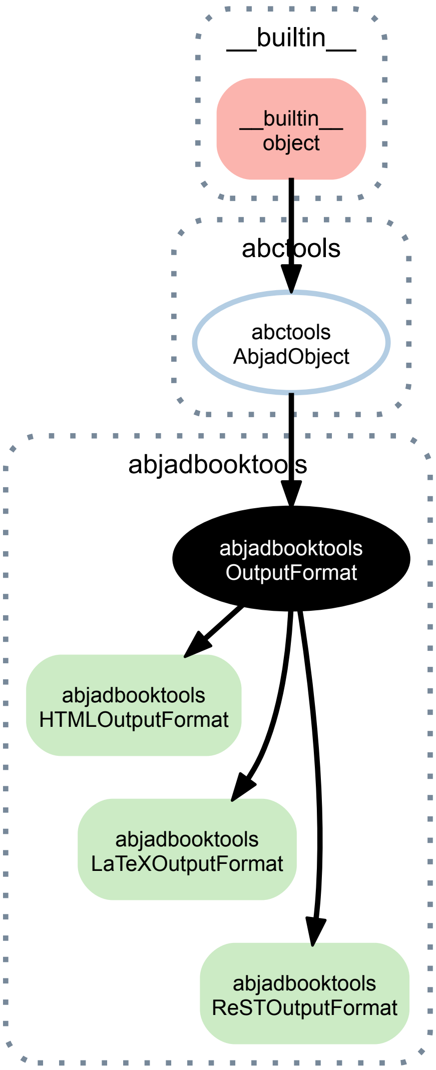
CHAPTER

FIFTYTWO

ABJADBOOKTOOLS

52.1 Abstract Classes

52.1.1 abjadbooktools.OutputFormat



class `abjadbooktools.`**`OutputFormat`**(*code_block_opening*, *code_block_closing*, *code_indent*, *image_block*, *image_format*)

Read-only properties

OutputFormat.**code_block_closing**

OutputFormat.**code_block_opening**

OutputFormat.**code_indent**

OutputFormat.**image_block**

OutputFormat.**image_format**

OutputFormat.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

OutputFormat.**__call__**(*code_block*, *image_dict*)

OutputFormat.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

OutputFormat.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OutputFormat.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

OutputFormat.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OutputFormat.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

OutputFormat.**__ne__**(*expr*)
Defined equal to the opposite of equality.

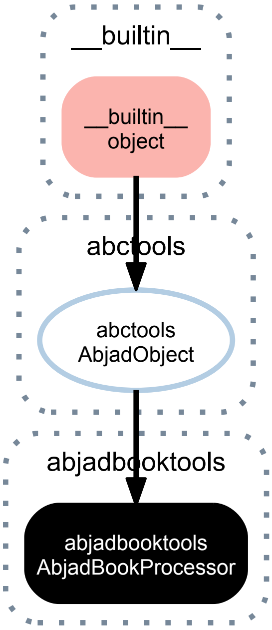
Return boolean.

OutputFormat.**__repr__**()
Interpreter representation of Abjad object.

Return string.

52.2 Concrete Classes

52.2.1 abjadbooktools.AbjadBookProcessor



class `abjadbooktools.`**`AbjadBookProcessor`**(*directory*, *lines*, *output_format*, *skip_rendering=False*, *image_prefix='image'*, *verbose=False*)

Read-only properties

`AbjadBookProcessor.`**`directory`**

`AbjadBookProcessor.`**`image_prefix`**

`AbjadBookProcessor.`**`lines`**

`AbjadBookProcessor.`**`output_format`**

`AbjadBookProcessor.`**`skip_rendering`**

`AbjadBookProcessor.`**`storage_format`**

Storage format of Abjad object.

Return string.

`AbjadBookProcessor.`**`verbose`**

Methods

`AbjadBookProcessor.`**`update_status`**(*line*)

Special methods

`AbjadBookProcessor.`**`__call__`**(*verbose=True*)

`AbjadBookProcessor.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

AbjadBookProcessor.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadBookProcessor.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjadBookProcessor.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadBookProcessor.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

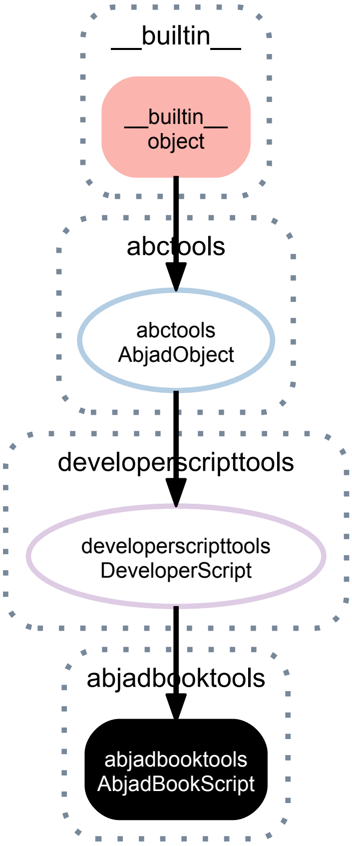
AbjadBookProcessor.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjadBookProcessor.**__repr__**()
Interpreter representation of Abjad object.

Return string.

52.2.2 abjadbooktools.AbjadBookScript



class abjadbooktools.**AbjadBookScript**

Read-only properties

AbjadBookScript.**alias**

AbjadBookScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

AbjadBookScript.**formatted_help**

AbjadBookScript.**formatted_usage**

AbjadBookScript.**formatted_version**

AbjadBookScript.**long_description**

AbjadBookScript.**output_formats**

AbjadBookScript.**program_name**
The name of the script, callable from the command line.

AbjadBookScript.**scripting_group**
The script's scripting subcommand group.

AbjadBookScript.**short_description**

AbjadBookScript.**storage_format**
Storage format of Abjad object.

Return string.

AbjadBookScript.**version**

Methods

AbjadBookScript.**process_args**(*args*)

AbjadBookScript.**setup_argument_parser**(*parser*)

Special methods

AbjadBookScript.**__call__**(*args=None*)

AbjadBookScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

AbjadBookScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadBookScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjadBookScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjadBookScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

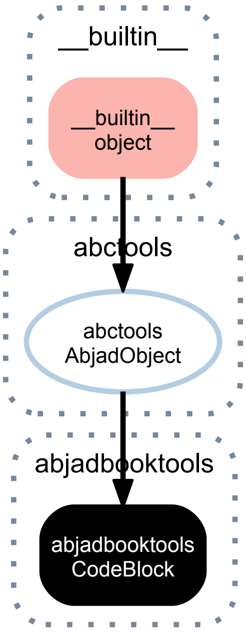
AbjadBookScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjadBookScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

52.2.3 abjadbooktools.CodeBlock



class abjadbooktools.**CodeBlock**(*lines*, *starting_line_number*, *ending_line_number*, *hide=False*, *strip_prompt=False*)

Read-only properties

CodeBlock.**ending_line_number**

CodeBlock.**hide**

CodeBlock.**lines**

CodeBlock.**processed_results**

CodeBlock.**starting_line_number**

CodeBlock.**storage_format**
Storage format of Abjad object.

Return string.

CodeBlock.**strip_prompt**

Methods

CodeBlock.**read**(*pipe*)

Special methods

`CodeBlock.__call__`(*processor*, *pipe*, *image_count=0*, *directory=None*, *image_prefix='image'*, *verbose=False*)

`CodeBlock.__eq__`(*expr*)

`CodeBlock.__ge__`(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`CodeBlock.__gt__`(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`CodeBlock.__le__`(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`CodeBlock.__lt__`(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`CodeBlock.__ne__`(*expr*)

Defined equal to the opposite of equality.

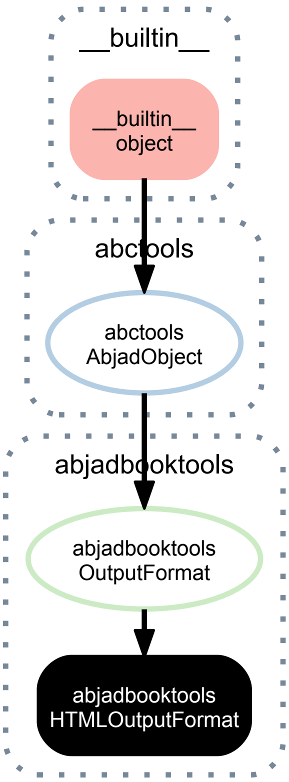
Return boolean.

`CodeBlock.__repr__`()

Interpreter representation of Abjad object.

Return string.

52.2.4 abjadbooktools.HTMLOutputFormat



class abjadbooktools.**HTMLOutputFormat**

Read-only properties

HTMLOutputFormat.**code_block_closing**

HTMLOutputFormat.**code_block_opening**

HTMLOutputFormat.**code_indent**

HTMLOutputFormat.**image_block**

HTMLOutputFormat.**image_format**

HTMLOutputFormat.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

HTMLOutputFormat.**__call__**(*code_block*, *image_dict*)

HTMLOutputFormat.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

HTMLOutputFormat.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HTMLOutputFormat.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

HTMLOutputFormat.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

HTMLOutputFormat.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

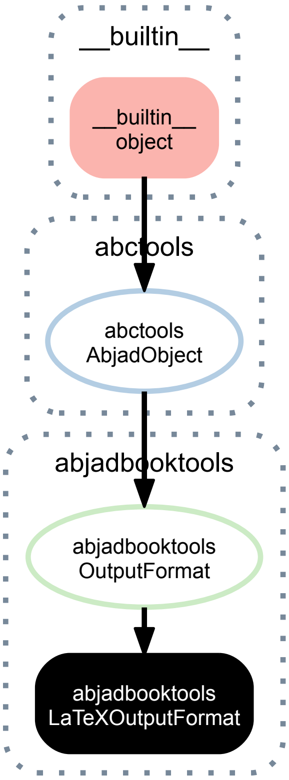
HTMLOutputFormat.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

HTMLOutputFormat.**__repr__**()
Interpreter representation of Abjad object.

Return string.

52.2.5 abjadbooktools.LaTeXOutputFormat



class `abjadbooktools.`**`LaTeXOutputFormat`**

Read-only properties

`LaTeXOutputFormat.`**`code_block_closing`**

`LaTeXOutputFormat.`**`code_block_opening`**

`LaTeXOutputFormat.`**`code_indent`**

`LaTeXOutputFormat.`**`image_block`**

`LaTeXOutputFormat.`**`image_format`**

`LaTeXOutputFormat.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`LaTeXOutputFormat.`**`__call__`**(*code_block*, *image_dict*)

`LaTeXOutputFormat.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`LaTeXOutputFormat.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LaTeXOutputFormat.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LaTeXOutputFormat.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LaTeXOutputFormat.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

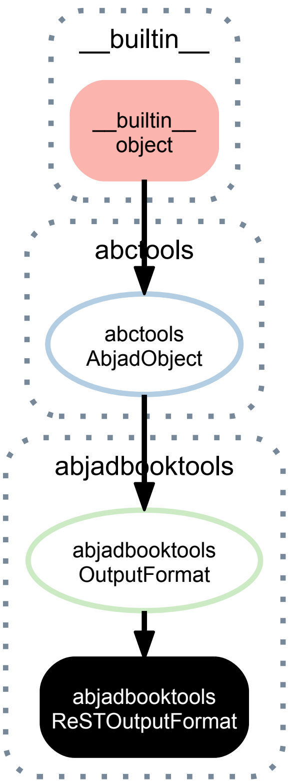
LaTeXOutputFormat.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LaTeXOutputFormat.**__repr__**()
Interpreter representation of Abjad object.

Return string.

52.2.6 abjadbooktools.ReSTOutputFormat



class abjadbooktools.**ReSTOutputFormat**

Read-only properties

ReSTOutputFormat.**code_block_closing**

ReSTOutputFormat.**code_block_opening**

ReSTOutputFormat.**code_indent**

ReSTOutputFormat.**image_block**

ReSTOutputFormat.**image_format**

ReSTOutputFormat.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

ReSTOutputFormat.**__call__**(*code_block*, *image_dict*)

ReSTOutputFormat.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ReSTOutputFormat.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOutputFormat.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTOutputFormat.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOutputFormat.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOutputFormat.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ReSTOutputFormat.**__repr__**()
Interpreter representation of Abjad object.

Return string.

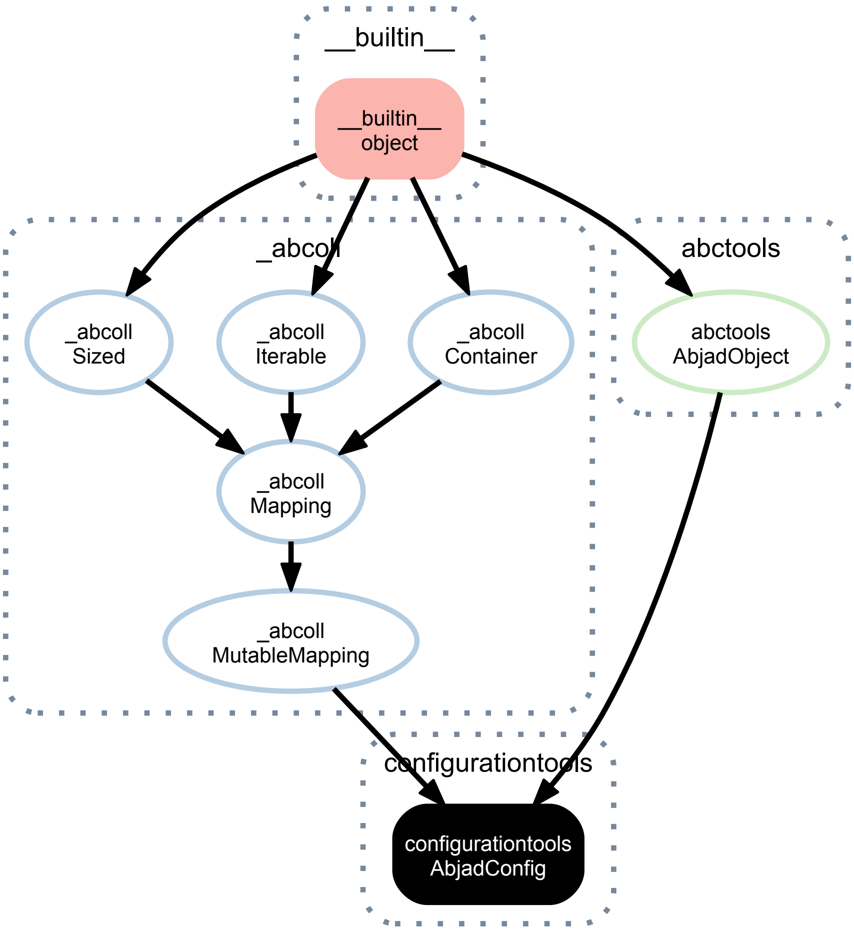
CHAPTER

FIFTYTHREE

CONFIGURATIONTOOLS

53.1 Concrete Classes

53.1.1 configurationtools.AbjadConfig



class `configurationtools.`**`AbjadConfig`**

Abjad configuration object:

```
>>> ABJCONFIG = configurationtools.AbjadConfig()
>>> ABJCONFIG['accidental_spelling']
'mixed'
```

On instantiation, *AbjadConfig* creates the *$HOME/.abjad/* directory if it does not already exist.

It then attempts to read an *abjad.cfg* file in that directory, parsing it as a *ConfigObj* configuration. A default configuration is generated if no file is found.

The *ConfigObj* instance is validated, and key:value pairs which fail validation are replaced by default values.

The configuration is then written back to disk.

Finally, the Abjad output directory is created if it does not already exist, by referencing the 'abjad_output' key in the configuration.

AbjadConfig supports the mutable mapping interface, and can be subscripted as a dictionary.

Returns *AbjadConfig* instance.

Read-only properties

`AbjadConfig.`**`ABJAD_CONFIG_DIRECTORY_PATH`**

`AbjadConfig.`**`ABJAD_CONFIG_FILE_PATH`**

`AbjadConfig.`**`ABJAD_EXPERIMENTAL_PATH`**

`AbjadConfig.`**`ABJAD_OUTPUT_PATH`**

`AbjadConfig.`**`ABJAD_PATH`**

`AbjadConfig.`**`ABJAD_ROOT_PATH`**

`AbjadConfig.`**`HOME_PATH`**

`AbjadConfig.`**`storage_format`**
Storage format of Abjad object.

Return string.

Methods

`AbjadConfig.`**`clear`**()

`AbjadConfig.`**`get`**(*key*, *default=None*)

`AbjadConfig.`**`get_config_spec`**()

`AbjadConfig.`**`get_initial_comment`**()

`AbjadConfig.`**`get_option_comments`**()

`AbjadConfig.`**`get_option_definitions`**()

`AbjadConfig.`**`get_option_specs`**()

`AbjadConfig.`**`items`**()

`AbjadConfig.`**`iteritems`**()

`AbjadConfig.`**`iterkeys`**()

`AbjadConfig.`**`itervalues`**()

`AbjadConfig.`**`keys`**()

`AbjadConfig.`**`pop`**(*key*, *default=<object object at 0x1002b0040>*)

`AbjadConfig.`**`popitem`**()

`AbjadConfig.`**`setdefault`**(*key*, *default=None*)

`AbjadConfig.`**`update`**(**args*, ***kwds*)

`AbjadConfig.`**`values`**()

Special methods

`AbjadConfig.__contains__(key)`

`AbjadConfig.__delitem__(i)`

`AbjadConfig.__eq__(other)`

`AbjadConfig.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjadConfig.__getitem__(i)`

`AbjadConfig.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`AbjadConfig.__iter__()`

`AbjadConfig.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjadConfig.__len__()`

`AbjadConfig.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjadConfig.__ne__(other)`

`AbjadConfig.__repr__()`
Interpreter representation of Abjad object.

Return string.

`AbjadConfig.__setitem__(i, arg)`

53.2 Functions

53.2.1 configurationtools.get_abjad_revision_string

`configurationtools.get_abjad_revision_string()`
New in version 2.0. Get Abjad revision string:

```
>>> configurationtools.get_abjad_revision_string()
'5280'
```

Return string.

53.2.2 configurationtools.get_abjad_startup_string

`configurationtools.get_abjad_startup_string()`

53.2.3 configurationtools.get_abjad_version_string

`configurationtools.get_abjad_version_string()`
New in version 2.0. Get Abjad version string:

```
>>> configurationtools.get_abjad_version_string()
'2.12'
```

Return string.

53.2.4 configurationtools.get_lilypond_version_string

configurationtools.**get_lilypond_version_string**()

New in version 2.0. Get LilyPond version string:

```
>>> configurationtools.get_lilypond_version_string()
'2.13.61'
```

Return string.

53.2.5 configurationtools.get_python_version_string

configurationtools.**get_python_version_string**()

New in version 2.0. Get Python version string:

```
>>> configurationtools.get_python_version_string()
'2.6.1'
```

Return string.

53.2.6 configurationtools.get_tab_width

configurationtools.**get_tab_width**()

New in version 2.9. Get system tab width:

```
>>> configurationtools.get_tab_width()
4
```

The value is used by various functions that generate or test code in the system.

Return nonnegative integer.

53.2.7 configurationtools.get_text_editor

configurationtools.**get_text_editor**()

New in version 2.2. Get OS-appropriate text editor.

53.2.8 configurationtools.list_abjad_environment_variables

configurationtools.**list_abjad_environment_variables**()

New in version 1.1. List Abjad environment variables.

Return tuple of zero or more environment variable / setting pairs.

Abjad environment variables are defined in `abjad/tools/configurationtools/AbjadConfig/AbjadConfig.`

53.2.9 configurationtools.list_package_dependency_versions

configurationtools.**list_package_dependency_versions**()

List package dependency versions:

```
>>> configurationtools.list_package_dependency_versions()
{'sphinx': '1.1.2', 'py.test': '2.1.2'}
```

Return dictionary.

53.2.10 configurationtools.read_abjad_user_config_file

configurationtools.**read_abjad_user_config_file**(*attribute_name*)

Read the content of the config file `$HOME/.abjad/config.py`.

Returns a dictionary of var : value entries.

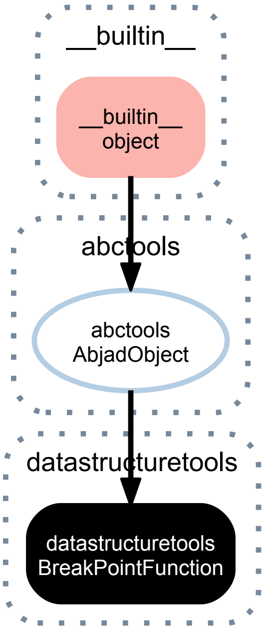
CHAPTER

FIFTYFOUR

DATASTRUCTURETOOLS

54.1 Concrete Classes

54.1.1 datastructuretools.BreakPointFunction



class datastructuretools.**BreakPointFunction**(*bpf*)

A break-point function:

```
>>> bpf = datastructuretools.BreakPointFunction({
...     0.:   0.,
...     0.75: (-1, 1.),
...     1.:   0.25,
... })
```

Allows interpolated lookup, and supports discontiguities on the y-axis:

```
>>> for x in (-0.5, 0., 0.25, 0.5, 0.7499, 0.75, 1., 1.5):
...     bpf[x]
0.0
0.0
-0.3333333333333333
-0.6666666666666666
-0.9998666666666667
1.0
0.25
0.25
```

Return *BreakPointFunction* instance.

Read-only properties

`BreakPointFunction.`**`bpf`**

A copy of the BreakPointFunction's internal data-structure:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).bpf
{0.0: (0.25,), 0.5: (1.3,), 1.0: (0.9,)}
```

Return dict.

`BreakPointFunction.`**`dc_bias`**

The mean y-value of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 0.25: 0., 0.5: (0.75, 0.25), 1.: 1.}
...     ).dc_bias
0.4
```

Return number.

`BreakPointFunction.`**`gnuplot_format`**

`BreakPointFunction.`**`storage_format`**

Storage format of Abjad object.

Return string.

`BreakPointFunction.`**`x_center`**

The arithmetic mean of a BreakPointFunction's x-range:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_center
0.5
```

Return number.

`BreakPointFunction.`**`x_range`**

The minimum and maximum x-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_range
(0.0, 1.0)
```

Return pair.

`BreakPointFunction.`**`x_values`**

The sorted x-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).x_values
(0.0, 0.5, 1.0)
```

Return tuple.

`BreakPointFunction.`**`y_center`**

The arithmetic mean of a BreakPointFunction's y-range:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).y_center
0.775
```

Return number.

`BreakPointFunction.`**`y_range`**

The minimum and maximum y-values of a BreakPointFunction:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0.25, 0.5: 1.3, 1.: 0.9}).y_range
(0.25, 1.3)
```

Return pair.

Methods

`BreakPointFunction.`**`clip_x_axis`**(*minimum=0*, *maximum=1*)

Clip x-axis between *minimum* and *maximum*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf.clip_x_axis(minimum=0.25, maximum=0.75)
BreakPointFunction({
    0.25: (0.75,),
    0.75: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`clip_y_axis`**(*minimum=0*, *maximum=1*)

Clip y-axis between *minimum* and *maximum*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf.clip_y_axis(minimum=0.25, maximum=0.75)
BreakPointFunction({
    0.0: (0.75,),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`concatenate`**(*expr*)

Concatenate self with *expr*:

```
>>> one = datastructuretools.BreakPointFunction(
...     {0.0: 0.0, 1.0: 1.0})
>>> two = datastructuretools.BreakPointFunction(
...     {0.5: 0.75, 1.5: 0.25})
```

```
>>> one.concatenate(two)
BreakPointFunction({
    0.0: (0.0,),
    1.0: (1.0, 0.75),
    2.0: (0.25,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`get_y_at_x`**(*x*)

Get *y*-value at *x*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 1.), 1.: 0.5})
```

```
>>> bpf.get_y_at_x(-1000)
0.0
```

```
>>> bpf.get_y_at_x(0.)
0.0
```

```
>>> bpf.get_y_at_x(0.25)
-0.5
```

```
>>> bpf.get_y_at_x(0.4999)
-0.9998
```

```
>>> bpf.get_y_at_x(0.5)
1.0
```

```
>>> bpf.get_y_at_x(0.75)
0.75
```

```
>>> bpf.get_y_at_x(1.)
0.5
```

```
>>> bpf.get_y_at_x(1000)
0.5
```

Return Number.

`BreakPointFunction.`**`invert`**`(`*y_center=None*`)`

Invert self:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 1.: 1.}).invert()
BreakPointFunction({
    0.0: (1.0,),
    1.0: (0.0,)
})
```

If *y_center* is not None, use *y_center* as the axis of inversion:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 1.: 1.}).invert(0)
BreakPointFunction({
    0.0: (0.0,),
    1.0: (-1.0,)
})
```

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 1.: 1.}).invert(0.25)
BreakPointFunction({
    0.0: (0.5,),
    1.0: (-0.5,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`normalize_axes`**`()`

Scale both x and y axes between 0 and 1:

```
>>> datastructuretools.BreakPointFunction(
...     {0.25: 0.25, 0.75: 0.75}).normalize_axes()
BreakPointFunction({
    0.0: (0.0,),
    1.0: (1.0,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`reflect`**`(`*x_center=None*`)`

Reflect x values of a *BreakPointFunction*:

```
>>> datastructuretools.BreakPointFunction(
...     {0.25: 0., 0.5: (-1., 2.), 1: 1.}).reflect()
BreakPointFunction({
    0.25: (1.0,),
    0.75: (2.0, -1.0),
    1.0: (0.0,)
})
```

If *x_center* is not None, reflection will take *x_center* as the axis of reflection:

```
>>> datastructuretools.BreakPointFunction(
...     {0.25: 0., 0.5: (-1., 2.), 1: 1.}).reflect(x_center=0.25)
BreakPointFunction({
    -0.5: (1.0,),
    0.0: (2.0, -1.0),
    0.25: (0.0,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`remove_dc_bias`**`()`

Remove dc-bias from a *BreakPointFunction*:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 1.: 1.}).remove_dc_bias()
BreakPointFunction({
    0.0: (-0.5,),
    1.0: (0.5,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`scale_x_axis`**`(`*minimum=0*, *maximum=1*`)`

Scale x-axis between *minimum* and *maximum*:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 2.), 1.: 1.}
...     ).scale_x_axis(-2, 2)
BreakPointFunction({
    -2.0: (0.0,),
    0.0: (-1.0, 2.0),
    2.0: (1.0,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`scale_y_axis`**`(`*minimum=0*, *maximum=1*`)`

Scale y-axis between *minimum* and *maximum*:

```
>>> datastructuretools.BreakPointFunction(
...     {0.: 0., 0.5: (-1., 2.), 1.: 1.}
...     ).scale_y_axis(-2, 4)
BreakPointFunction({
    0.0: (0.0,),
    0.5: (-2.0, 4.0),
    1.0: (2.0,)
})
```

Emit new *BreakPointFunction* instance.

`BreakPointFunction.`**`set_y_at_x`**`(`*x*, *y*`)`

Set *y*-value at *x*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.0: 0.0, 1.0: 1.0})
```

With a number:

```
>>> bpf.set_y_at_x(0.25, 0.75)
>>> bpf
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.75,),
    1.0: (1.0,)
})
```

With a pair:

```
>>> bpf.set_y_at_x(0.6, (-2., 2.))
>>> bpf
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.75,),
    0.6: (-2.0, 2.0),
    1.0: (1.0,)
})
```

Delete all values at *x* when *y* is None:

```
>>> bpf.set_y_at_x(0., None)
>>> bpf
BreakPointFunction({
    0.25: (0.75,),
    0.6: (-2.0, 2.0),
    1.0: (1.0,)
})
```

Operate in place and return None.

`BreakPointFunction.`**`tessalate_by_ratio`**(*ratio*, *invert_on_negative=False*, *reflect_on_negative=False*, *y_center=None*)

Concatenate copies of a BreakPointFunction, stretched by the weights in *ratio*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.25: 0.9, 1.: 1.})
```

```
>>> bpf.tessalate_by_ratio((1, 2, 3))
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.9,),
    1.0: (1.0, 0.0),
    1.5: (0.9,),
    3.0: (1.0, 0.0),
    3.75: (0.9,),
    6.0: (1.0,)
})
```

Negative ratio values are still treated as weights.

If *invert_on_negative* is True, copies corresponding to negative ratio values will be inverted:

```
>>> bpf.tessalate_by_ratio((1, -2, 3), invert_on_negative=True)
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.9,),
    1.0: (1.0,),
    1.5: (0.09999999999999998,),
    3.0: (0.0,),
    3.75: (0.9,),
    6.0: (1.0,)
})
```

If *y_center* is not None, inversion will take *y_center* as the axis of inversion:

```
>>> bpf.tessalate_by_ratio((1, -2, 3), invert_on_negative=True, y_center=0)
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.9,),
    1.0: (1.0, 0.0),
    1.5: (-0.9,),
    3.0: (-1.0, 0.0),
    3.75: (0.9,),
    6.0: (1.0,)
})
```

If *reflect_on_negative* is True, copies corresponding to negative ratio values will be reflectd:

```
>>> bpf.tessalate_by_ratio((1, -2, 3), reflect_on_negative=True)
BreakPointFunction({
    0.0: (0.0,),
    0.25: (0.9,),
    1.0: (1.0,),
    2.5: (0.9,),
    3.0: (0.0,),
    3.75: (0.9,),
    6.0: (1.0,)
})
```

Inversion may be combined reflecting.

Emit new *BreakPointFunction* instance.

Special methods

BreakPointFunction.**__add__**(*expr*)
Add *expr* to all y-values in self:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf + 0.3
BreakPointFunction({
    0.0: (0.3,),
    0.75: (-0.7, 1.3),
    1.0: (0.55,)
})
```

expr may also be a BreakPointFunction instance:

```
>>> bpf2 = datastructuretools.BreakPointFunction({0.: 1., 1.: 0.})
>>> bpf + bpf2
BreakPointFunction({
    0.0: (1.0,),
    0.75: (-0.75, 1.25),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction*.

BreakPointFunction.**__div__**(*expr*)
Divide y-values in self by *expr*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf / 2.
BreakPointFunction({
    0.0: (0.0,),
    0.75: (-0.5, 0.5),
    1.0: (0.125,)
})
```

expr may also be a *BreakPointFunction* instance.

Emit new *BreakPointFunction*.

BreakPointFunction.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

BreakPointFunction.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BreakPointFunction.**__getitem__**(*item*)
Aliases BreakPointFunction.get_y_at_x().

BreakPointFunction.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BreakPointFunction.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BreakPointFunction.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BreakPointFunction.**__mul__**(*expr*)

Multiply y-values in self by *expr*:

```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf * 2.
BreakPointFunction({
    0.0: (0.0,),
    0.75: (-2.0, 2.0),
    1.0: (0.5,)
})
```

expr may also be a *BreakPointFunction* instance.

Emit new *BreakPointFunction*.

BreakPointFunction.**__ne__**(*expr*)

Defined equal to the opposite of equality.

Return boolean.

BreakPointFunction.**__repr__**()

BreakPointFunction.**__sub__**(*expr*)

Subtract *expr* from all y-values in self:

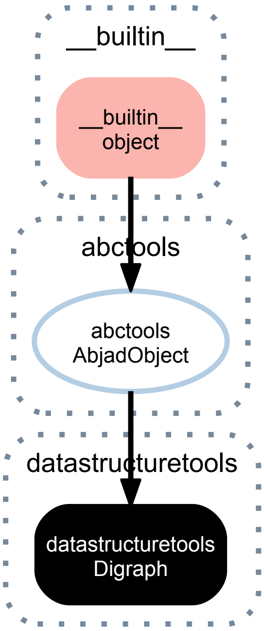
```
>>> bpf = datastructuretools.BreakPointFunction(
...     {0.: 0., 0.75: (-1., 1.), 1.: 0.25})
>>> bpf - 0.3
BreakPointFunction({
    0.0: (-0.3,),
    0.75: (-1.3, 0.7),
    1.0: (-0.04999999999999999,)
})
```

expr may also be a BreakPointFunction instance:

```
>>> bpf2 = datastructuretools.BreakPointFunction(
...     {0.: 1., 1.: 0.})
>>> bpf - bpf2
BreakPointFunction({
    0.0: (-1.0,),
    0.75: (-1.25, 0.75),
    1.0: (0.25,)
})
```

Emit new *BreakPointFunction*.

54.1.2 datastructuretools.Digraph



class `datastructuretools.`**`Digraph`**`(`*edges=None*`)`

A digraph, built out of edges - pairs of hashable objects:

```
>>> edges = [('a', 'b'), ('a', 'c'), ('a', 'f'), ('c', 'd'), ('d', 'e'), ('e', 'c')]
>>> digraph = datastructuretools.Digraph(edges)
>>> digraph
Digraph(edges=[('a', 'c'), ('a', 'b'), ('a', 'f'), ('c', 'd'), ('d', 'e'), ('e', 'c')])
```

```
>>> digraph.root_nodes
('a',)
>>> digraph.terminal_nodes
('b', 'f')
>>> digraph.cyclic_nodes
('c', 'd', 'e')
>>> digraph.is_cyclic
True
```

Return *Digraph* instance.

Read-only properties

`Digraph.`**`child_graph`**

A dictionary representation of the digraph where the keys are child nodes, and where each value is the set of that child's parents.

`Digraph.`**`cyclic_nodes`**

A tuple of those nodes which partake in a cycle.

`Digraph.`**`edges`**

A tuple of all edges in the graph.

`Digraph.`**`is_cyclic`**

Return True if the digraph contains any cycles.

`Digraph.`**`nodes`**

A tuple of all nodes in the graph.

`Digraph.`**`parent_graph`**

A dictionary representation of the digraph where the keys are parent nodes, and where each value is the set of that parent's children.

`Digraph.`**`root_nodes`**
A tuple of those nodes which have no parents.

`Digraph.`**`storage_format`**
Storage format of Abjad object.

Return string.

`Digraph.`**`terminal_nodes`**
A tuple of those nodes which have no children.

Methods

`Digraph.`**`partition`**`()`
Partition the digraph into a list of digraphs according to connectivity:

```
>>> edges = [('a', 'b'), ('a', 'c'), ('b', 'c'), ('b', 'd'), ('d', 'e')]
>>> edges.extend([('f', 'h'), ('g', 'h')])
>>> edges.append(('i', 'j'))
>>> digraph = datastructuretools.Digraph(edges)
>>> for graph in digraph.partition(): graph
...
Digraph(edges=[('a', 'c'), ('a', 'b'), ('b', 'c'), ('b', 'd'), ('d', 'e')])
Digraph(edges=[('f', 'h'), ('g', 'h')])
Digraph(edges=[('i', 'j')])
```

Return list of *Digraph* instances.

`Digraph.`**`reverse`**`()`
Reverse all edges in the graph:

```
>>> edges = [('a', 'b'), ('b', 'c'), ('b', 'd')]
>>> digraph = datastructuretools.Digraph(edges)
>>> digraph
Digraph(edges=[('a', 'b'), ('b', 'c'), ('b', 'd')])
```

```
>>> digraph.reverse()
Digraph(edges=[('b', 'a'), ('c', 'b'), ('d', 'b')])
```

Return *Digraph* instance.

Special methods

`Digraph.`**`__eq__`**`(`*expr*`)`

`Digraph.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Digraph.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Digraph.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Digraph.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

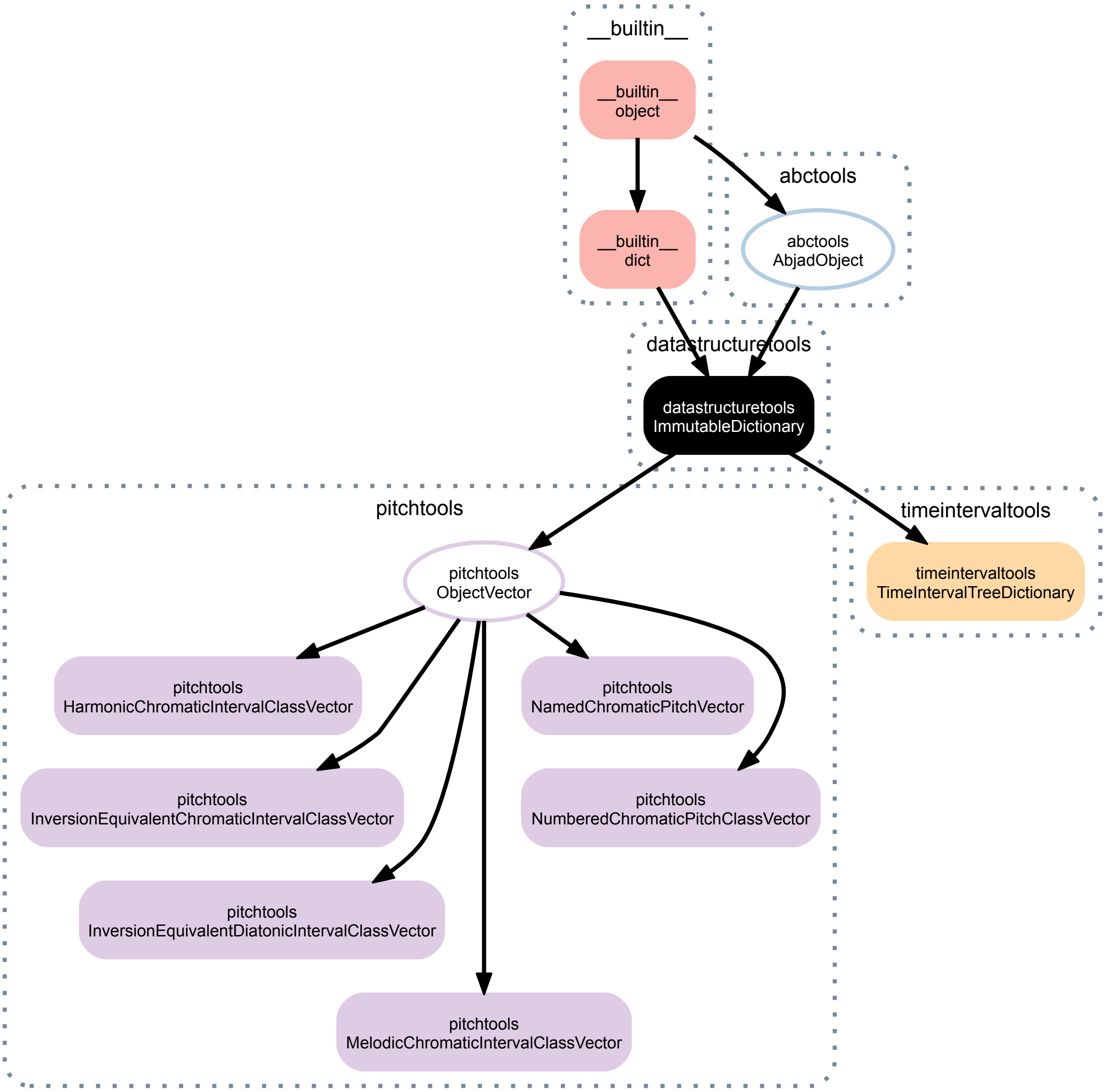
`Digraph.`**`__ne__`**`(`*expr*`)`

`Digraph.__repr__()`

Interpreter representation of Abjad object.

Return string.

54.1.3 datastructuretools.ImmutableDictionary



class `datastructuretools.ImmutableDictionary`

New in version 2.0. Immutable dictionary:

```
>>> dictionary = datastructuretools.ImmutableDictionary({'color': 'red', 'number': 9})
```

```
>>> dictionary
{'color': 'red', 'number': 9}
```

```
>>> dictionary['color']
'red'
```

```
>>> dictionary.size = 'large'
AttributeError: ImmutableDictionary objects are immutable.
```

```
>>> dictionary['size'] = 'large'
AttributeError: ImmutableDictionary objects are immutable.
```

Return immutable dictionary.

Read-only properties

`ImmutableDictionary.storage_format`
Storage format of Abjad object.

Return string.

Methods

`ImmutableDictionary.clear()` → None. Remove all items from D.

`ImmutableDictionary.copy()` → a shallow copy of D

`ImmutableDictionary.get(k[, d])` → D[k] if k in D, else d. d defaults to None.

`ImmutableDictionary.has_key(k)` → True if D has a key k, else False

`ImmutableDictionary.items()` → list of D's (key, value) pairs, as 2-tuples

`ImmutableDictionary.iteritems()` → an iterator over the (key, value) items of D

`ImmutableDictionary.iterkeys()` → an iterator over the keys of D

`ImmutableDictionary.itervalues()` → an iterator over the values of D

`ImmutableDictionary.keys()` → list of D's keys

`ImmutableDictionary.pop(k[, d])` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

`ImmutableDictionary.popitem()` → (k, v), remove and return some (key, value) pair as a 2-tuple; but raise KeyError if D is empty.

`ImmutableDictionary.setdefault(k[, d])` → D.get(k,d), also set D[k]=d if k not in D

`ImmutableDictionary.update([E], **F)` → None. Update D from dict/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

`ImmutableDictionary.values()` → list of D's values

`ImmutableDictionary.viewitems()` → a set-like object providing a view on D's items

`ImmutableDictionary.viewkeys()` → a set-like object providing a view on D's keys

`ImmutableDictionary.viewvalues()` → an object providing a view on D's values

Special methods

`ImmutableDictionary.__cmp__(y)` <==> *cmp(x, y)*

`ImmutableDictionary.__contains__(k)` → True if D has a key k, else False

`ImmutableDictionary.__delitem__(*args)`

`ImmutableDictionary.__eq__()`
x.__eq__(y) <==> x==y

`ImmutableDictionary.__ge__()`
x.__ge__(y) <==> x>=y

ImmutableDictionary.**__getitem__**()
x.__getitem__(y) <==> x[y]

ImmutableDictionary.**__gt__**()
x.__gt__(y) <==> x>y

ImmutableDictionary.**__iter__**() <==> *iter(x)*

ImmutableDictionary.**__le__**()
x.__le__(y) <==> x<=y

ImmutableDictionary.**__len__**() <==> *len(x)*

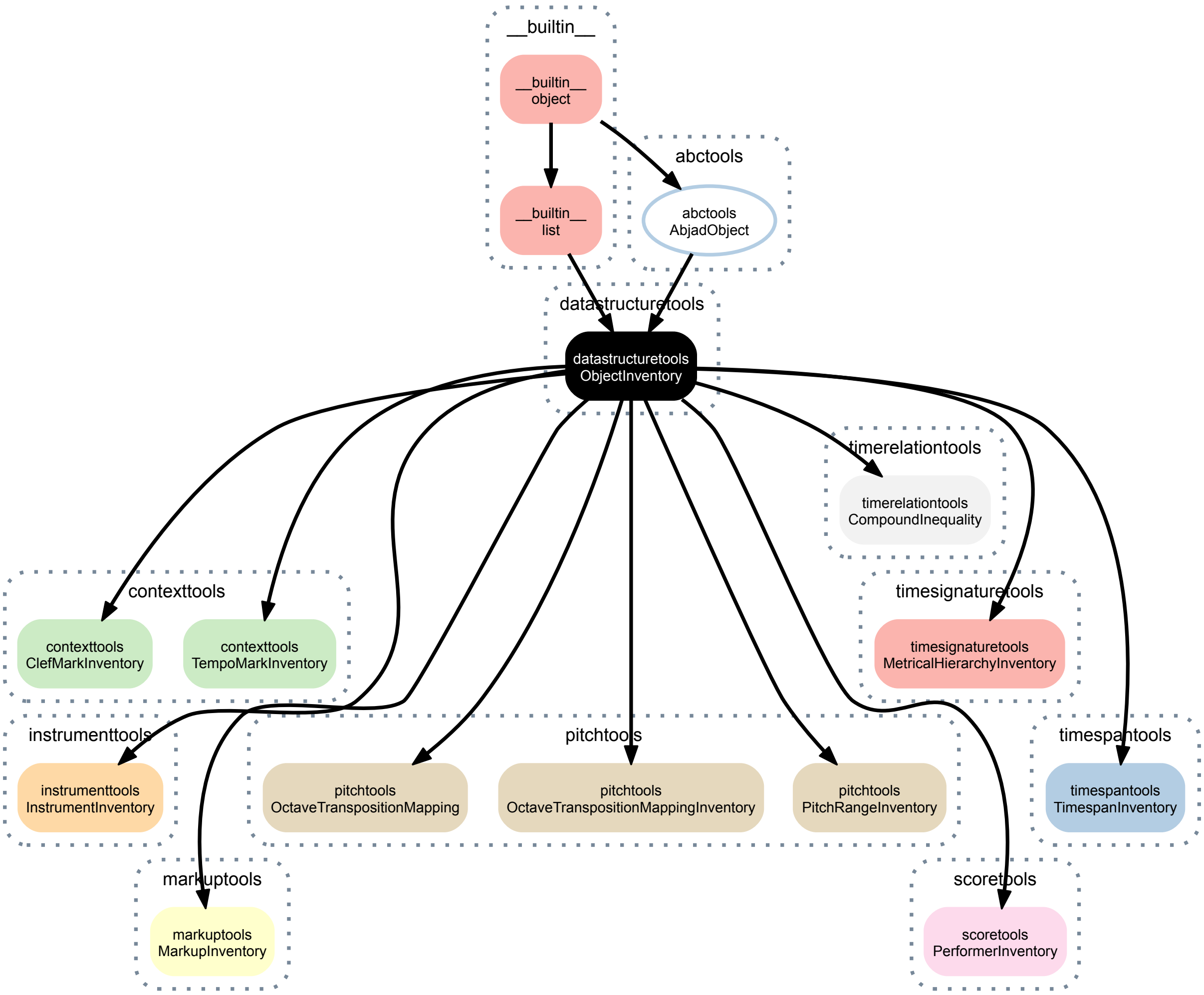
ImmutableDictionary.**__lt__**()
x.__lt__(y) <==> x<y

ImmutableDictionary.**__ne__**()
x.__ne__(y) <==> x!=y

ImmutableDictionary.**__repr__**() <==> *repr(x)*

ImmutableDictionary.**__setitem__**(**args*)

54.1.4 datastructuretools.ObjectInventory



class datastructuretools.**ObjectInventory**(*tokens=None*, *name=None*)
New in version 2.8. Ordered collection of custom objects.

Object inventories extend append(), extend() and __contains__() and allow token input.

Object inventories inherit from list and are mutable.

This class is an abstract base class that can not instantiate and should be subclassed.

Read-only properties

`ObjectInventory.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`ObjectInventory.`**`name`**
Read / write name of inventory.

Methods

`ObjectInventory.`**`append`**`(`*token*`)`
Change *token* to item and append.

`ObjectInventory.`**`count`**`(`*value*`)` → integer – return number of occurrences of value

`ObjectInventory.`**`extend`**`(`*tokens*`)`
Change *tokens* to items and extend.

`ObjectInventory.`**`index`**`(`*value*[, *start*[, *stop*]]`)` → integer – return first index of value.
Raises ValueError if the value is not present.

`ObjectInventory.`**`insert`**`()`
L.insert(index, object) – insert object before index

`ObjectInventory.`**`pop`**`(`[*index*]`)` → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

`ObjectInventory.`**`remove`**`()`
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

`ObjectInventory.`**`reverse`**`()`
L.reverse() – reverse *IN PLACE*

`ObjectInventory.`**`sort`**`()`
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

Special methods

`ObjectInventory.`**`__add__`**`()`
x.__add__(y) <==> x+y

`ObjectInventory.`**`__contains__`**`(`*token*`)`

`ObjectInventory.`**`__delitem__`**`()`
x.__delitem__(y) <==> del x[y]

`ObjectInventory.`**`__delslice__`**`()`
x.__delslice__(i, j) <==> del x[i:j]

Use of negative indices is not supported.

`ObjectInventory.`**`__eq__`**`()`
x.__eq__(y) <==> x==y

`ObjectInventory.`**`__ge__`**`()`
x.__ge__(y) <==> x>=y

`ObjectInventory.__getitem__()`
x.__getitem__(y) <==> x[y]

`ObjectInventory.__getslice__()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ObjectInventory.__gt__()`
x.__gt__(y) <==> x>y

`ObjectInventory.__iadd__()`
x.__iadd__(y) <==> x+=y

`ObjectInventory.__imul__()`
x.__imul__(y) <==> x*=y

`ObjectInventory.__iter__()` <==> *iter(x)*

`ObjectInventory.__le__()`
x.__le__(y) <==> x<=y

`ObjectInventory.__len__()` <==> *len(x)*

`ObjectInventory.__lt__()`
x.__lt__(y) <==> x<y

`ObjectInventory.__mul__()`
x.__mul__(n) <==> x*n

`ObjectInventory.__ne__()`
x.__ne__(y) <==> x!=y

`ObjectInventory.__repr__()`

`ObjectInventory.__reversed__()`
L.__reversed__() – return a reverse iterator over the list

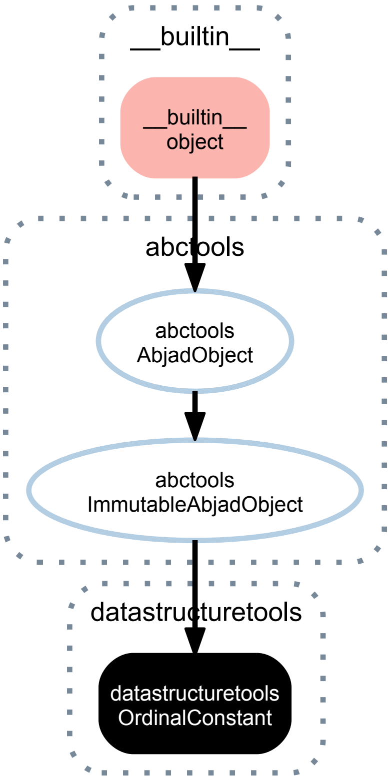
`ObjectInventory.__rmul__()`
x.__rmul__(n) <==> n*x

`ObjectInventory.__setitem__()`
x.__setitem__(i, y) <==> x[i]=y

`ObjectInventory.__setslice__()`
x.__setslice__(i, j, y) <==> x[i:j]=y

Use of negative indices is not supported.

54.1.5 datastructuretools.OrdinalConstant



class datastructuretools.**OrdinalConstant**(**args*, ***kwargs*)

New in version 1.0. Ordinal constant.

Initialize with *dimension*, *value* and *representation*:

```
>>> Left = datastructuretools.OrdinalConstant('x', -1, 'Left')
>>> Left
Left
```

```
>>> Right = datastructuretools.OrdinalConstant('x', 1, 'Right')
>>> Right
Right
```

```
>>> Left < Right
True
```

Comparing like-dimensioned ordinal constants is allowed:

```
>>> Up = datastructuretools.OrdinalConstant('y', 1, 'Up')
>>> Up
Up
```

```
>>> Down = datastructuretools.OrdinalConstant('y', -1, 'Down')
>>> Down
Down
```

```
>>> Down < Up
True
```

Comparing differently dimensioned ordinal constants raises an exception:

```
>>> import py.test
```

```
>>> bool(py.test.raises(Exception, 'Left < Up'))
True
```

The `Left`, `Right`, `Center`, `Up` and `Down` constants shown here load into Python's built-in namespace on Abjad import.

These four objects can be used as constant values supplied to keywords.

This behavior is similar to True, False and None.

Ordinal constants are immutable.

Read-only properties

`OrdinalConstant.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`OrdinalConstant.`**`__eq__`**`(`*expr*`)`

`OrdinalConstant.`**`__ge__`**`(`*expr*`)`

`OrdinalConstant.`**`__gt__`**`(`*expr*`)`

`OrdinalConstant.`**`__le__`**`(`*expr*`)`

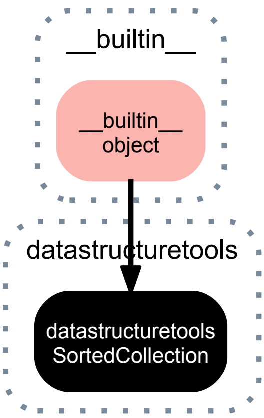
`OrdinalConstant.`**`__lt__`**`(`*expr*`)`

`OrdinalConstant.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`OrdinalConstant.`**`__repr__`**`()`

54.1.6 datastructuretools.SortedCollection



class `datastructuretools.`**`SortedCollection`**`(`*iterable=(), key=None*`)`
Sequence sorted by a key function.

SortedCollection() is much easier to work with than using bisect() directly. It supports key functions like those use in sorted(), min(), and max(). The result of the key function call is saved so that keys can be searched efficiently.

Instead of returning an insertion-point which can be hard to interpret, the five find-methods return a specific item in the sequence. They can scan for exact matches, the last item less-than-or-equal to a key, or the first item greater-than-or-equal to a key.

Once found, an item's ordinal position can be located with the index() method. New items can be added with the insert() and insert_right() methods. Old items can be deleted with the remove() method.

The usual sequence methods are provided to support indexing, slicing, length lookup, clearing, copying, forward and reverse iteration, contains checking, item counts, item removal, and a nice looking repr.

Finding and indexing are O(log n) operations while iteration and insertion are O(n). The initial sort is O(n log n).

The key function is stored in the 'key' attibute for easy introspection or so that you can assign a new key function (triggering an automatic re-sort).

In short, the class was designed to handle all of the common use cases for bisect but with a simpler API and support for key functions.

```
>>> from pprint import pprint
>>> from operator import itemgetter
```

```
>>> s = datastructuretools.SortedCollection(key=itemgetter(2))
>>> for record in [
...         ('roger', 'young', 30),
...         ('angela', 'jones', 28),
...         ('bill', 'smith', 22),
...         ('david', 'thomas', 32)]:
...     s.insert(record)
```

```
>>> pprint(list(s))         # show records sorted by age
[('bill', 'smith', 22),
 ('angela', 'jones', 28),
 ('roger', 'young', 30),
 ('david', 'thomas', 32)]
```

```
>>> s.find_le(29)           # find oldest person aged 29 or younger
('angela', 'jones', 28)
>>> s.find_lt(28)           # find oldest person under 28
('bill', 'smith', 22)
>>> s.find_gt(28)           # find youngest person over 28
('roger', 'young', 30)
```

```
>>> r = s.find_ge(32)       # find youngest person aged 32 or older
>>> s.index(r)              # get the index of their record
3
>>> s[3]                    # fetch the record at that index
('david', 'thomas', 32)
```

```
>>> s.key = itemgetter(0)   # now sort by first name
>>> pprint(list(s))
[('angela', 'jones', 28),
 ('bill', 'smith', 22),
 ('david', 'thomas', 32),
 ('roger', 'young', 30)]
```

Read/write properties

`SortedCollection.`**`key`**
key function

Methods

`SortedCollection.`**`clear`**`()`

`SortedCollection.`**`copy`**`()`

`SortedCollection.`**`count`**`(`*item*`)`
Return number of occurrences of item

SortedCollection.**find**(*k*)
Return first item with a key == k. Raise ValueError if not found.

SortedCollection.**find_ge**(*k*)
Return first item with a key >= equal to k. Raise ValueError if not found

SortedCollection.**find_gt**(*k*)
Return first item with a key > k. Raise ValueError if not found

SortedCollection.**find_le**(*k*)
Return last item with a key <= k. Raise ValueError if not found.

SortedCollection.**find_lt**(*k*)
Return last item with a key < k. Raise ValueError if not found.

SortedCollection.**index**(*item*)
Find the position of an item. Raise ValueError if not found.

SortedCollection.**insert**(*item*)
Insert a new item. If equal keys are found, add to the left

SortedCollection.**insert_right**(*item*)
Insert a new item. If equal keys are found, add to the right

SortedCollection.**remove**(*item*)
Remove first occurence of item. Raise ValueError if not found

Special methods

SortedCollection.**__contains__**(*item*)

SortedCollection.**__getitem__**(*i*)

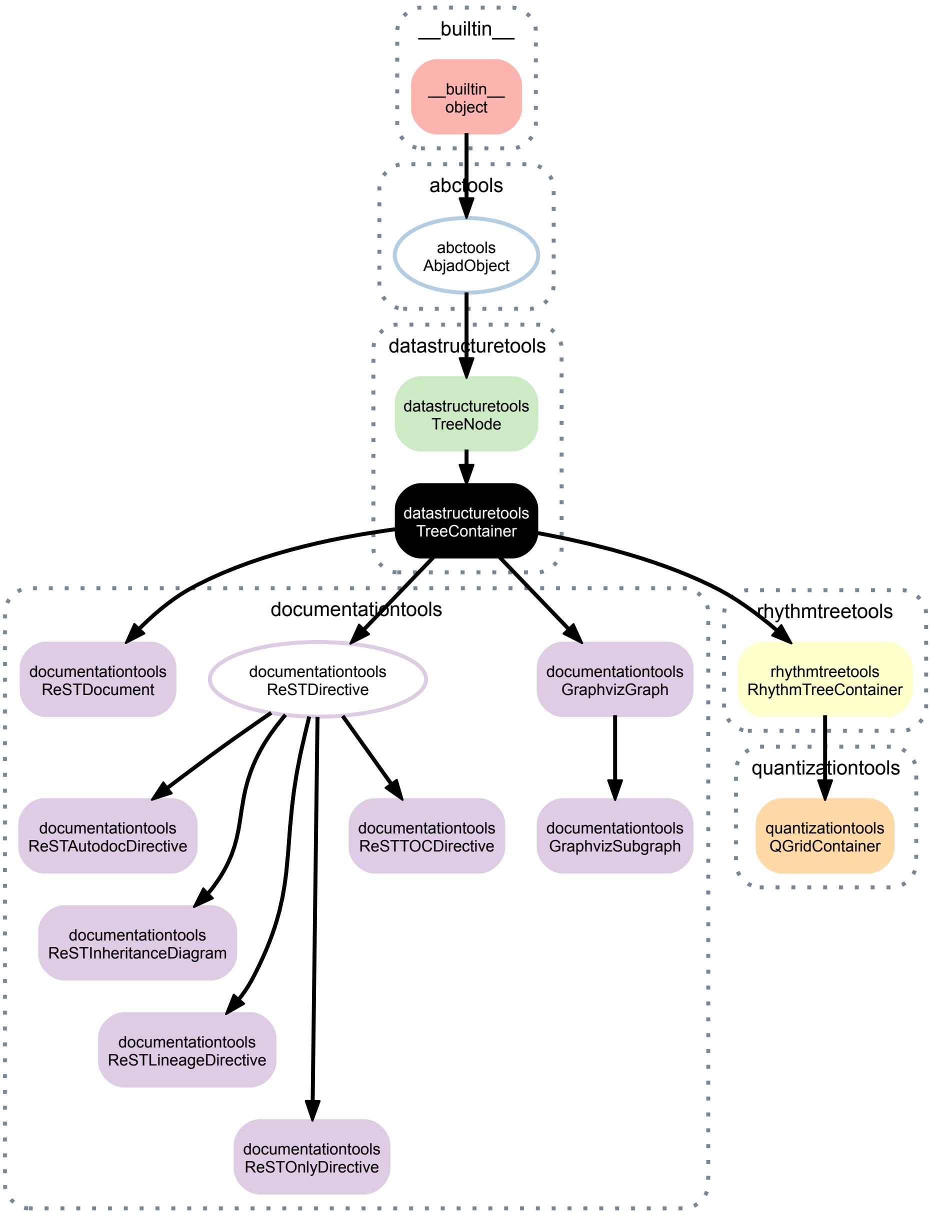
SortedCollection.**__iter__**()

SortedCollection.**__len__**()

SortedCollection.**__repr__**()

SortedCollection.**__reversed__**()

54.1.7 datastructuretools.TreeContainer



class `datastructuretools.`**`TreeContainer`**(*children=None*, *name=None*)

An inner node in a generalized tree data-structure:

```
>>> a = datastructuretools.TreeContainer()
>>> a
TreeContainer()
```

```
>>> b = datastructuretools.TreeNode()
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *TreeContainer* instance.

Read-only properties

`TreeContainer.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`TreeContainer.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`TreeContainer.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
```

```
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

TreeContainer.**graph_order**

TreeContainer.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

TreeContainer.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

TreeContainer.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`TreeContainer.`**`parent`**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`TreeContainer.`**`proper_parentage`**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

`TreeContainer.`**`root`**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

TreeContainer.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

TreeContainer.**name**

Methods

TreeContainer.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

TreeContainer.**extend**(*expr*)
Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
```

```
            TreeNode()
            )
        )
```

Return *None*.

`TreeContainer.`**`index`**`(`*node*`)`

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`TreeContainer.`**`insert`**`(`*i*, *node*`)`

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`TreeContainer.`**`pop`**`(`*i=-1*`)`

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

`TreeContainer.`**`remove`**`(`*node*`)`

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

`TreeContainer.`**`__contains__`**`(`*expr*`)`

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

`TreeContainer.`**`__copy__`**`(`**args*`)`

`TreeContainer.`**`__deepcopy__`**`(`**args*`)`

`TreeContainer.`**`__delitem__`**`(`*i*`)`

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

`TreeContainer.__eq__(expr)`
True if type, duration and children are equivalent, otherwise False.

Return boolean.

`TreeContainer.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TreeContainer.__getitem__(i)`
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

`TreeContainer.__getstate__()`

`TreeContainer.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`TreeContainer.__iter__()`

`TreeContainer.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TreeContainer.__len__()`
Return nonnegative integer number of nodes in container.

`TreeContainer.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`TreeContainer.__ne__(expr)`

`TreeContainer.__repr__()`

`TreeContainer.__setitem__(i, expr)`
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

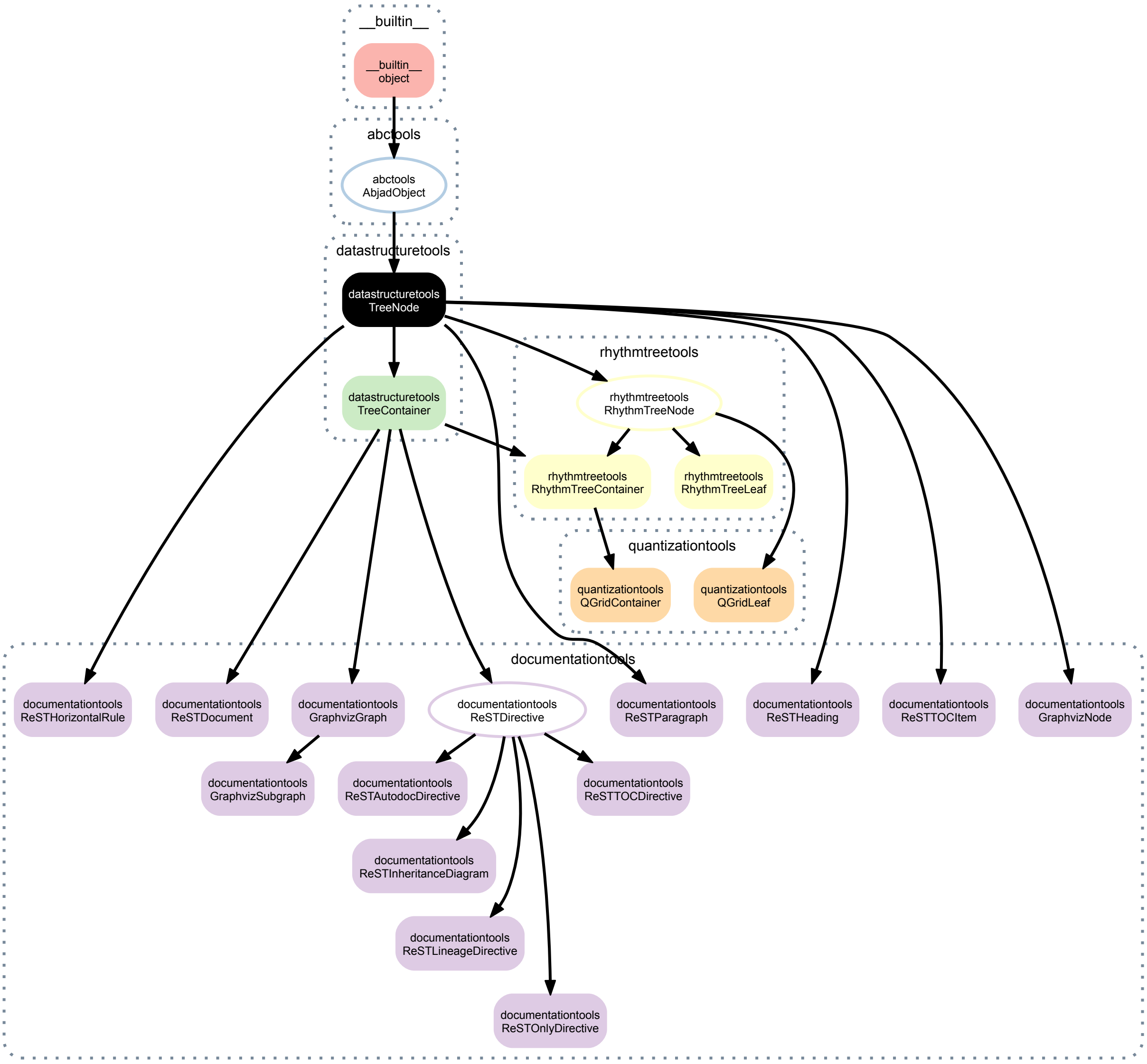
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

`TreeContainer.__setstate__(state)`

54.1.8 datastructuretools.TreeNode



class `datastructuretools.`**`TreeNode`**(*name=None*)

A node in a generalized tree.

Return *TreeNode* instance.

Read-only properties

`TreeNode.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

TreeNode.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

TreeNode.**graph_order**

TreeNode.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

TreeNode.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

TreeNode.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

TreeNode.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

TreeNode.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

TreeNode.**name**

Special methods

TreeNode.**__copy__**(*args)

TreeNode.**__deepcopy__**(*args)

TreeNode.**__eq__**(expr)

TreeNode.**__ge__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

TreeNode.**__getstate__**()

TreeNode.**__gt__**(expr)
Abjad objects by default do not implement this method.

Raise exception

TreeNode.**__le__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

TreeNode.**__lt__**(expr)
Abjad objects by default do not implement this method.

Raise exception.

TreeNode.**__ne__**(expr)

TreeNode.**__repr__**()

TreeNode.**__setstate__**(state)

CHAPTER

FIFTYFIVE

DECORATORTOOLS

55.1 Functions

55.1.1 decoratortools.requires

decoratortools.**requires**(**tests*)

New in version 2.6. Function decorator to require input parameter *tests*.

Example:

```
>>> @decoratortools.requires(
...     mathtools.is_nonnegative_integer, string)
>>> def multiply_string(n, string): return n * string
```

```
>>> multiply_string(2, 'bar')
'barbar'
```

```
>>> multiply_string(2.5, 'bar')
...
AssertionError: is_nonnegative_integer(2.5) does not return true.
```

Decorator target is available like this:

```
>>> multiply_string.func_closure[1].cell_contents
<function multiply_string at 0x104e512a8>
```

Decorator tests are available like this:

```
>>> multiply_string.func_closure[0].cell_contents
(<function is_nonnegative_integer at 0x104725d70>, <type 'str'>)
```

Return decorated function in the form of function wrapper.

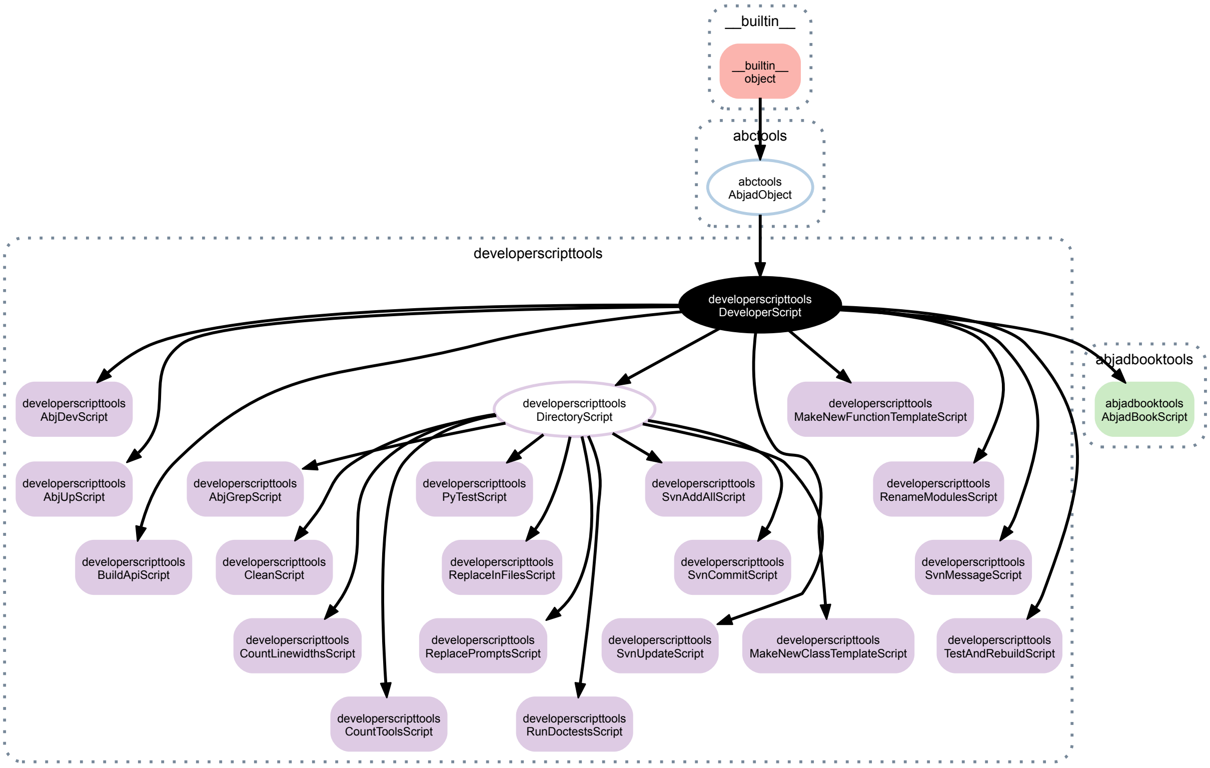
CHAPTER

FIFTYSIX

DEVELOPERSCRIPTTOOLS

56.1 Abstract Classes

56.1.1 developerscripttools.DeveloperScript



class `developerscripttools.`**`DeveloperScript`**

Abjad object-oriented model of a developer script.

DeveloperScript is the abstract parent from which concrete developer scripts inherit.

Developer scripts can be called from the command line, generally via the *ajv* command.

Developer scripts can be instantiated by other developer scripts in order to share functionality.

Read-only properties

`DeveloperScript.`**`alias`**

The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

`DeveloperScript.`**`argument_parser`**

The script's instance of argparse.ArgumentParser.

DeveloperScript.**formatted_help**

DeveloperScript.**formatted_usage**

DeveloperScript.**formatted_version**

DeveloperScript.**long_description**
The long description, printed after arguments explanations.

DeveloperScript.**program_name**
The name of the script, callable from the command line.

DeveloperScript.**scripting_group**
The script's scripting subcommand group.

DeveloperScript.**short_description**
The short description of the script, printed before arguments explanations.

Also used as a summary in other contexts.

DeveloperScript.**storage_format**
Storage format of Abjad object.

Return string.

DeveloperScript.**version**
The version number of the script.

Methods

DeveloperScript.**process_args**(*args*)

DeveloperScript.**setup_argument_parser**()

Special methods

DeveloperScript.**__call__**(*args=None*)

DeveloperScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DeveloperScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DeveloperScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DeveloperScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DeveloperScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

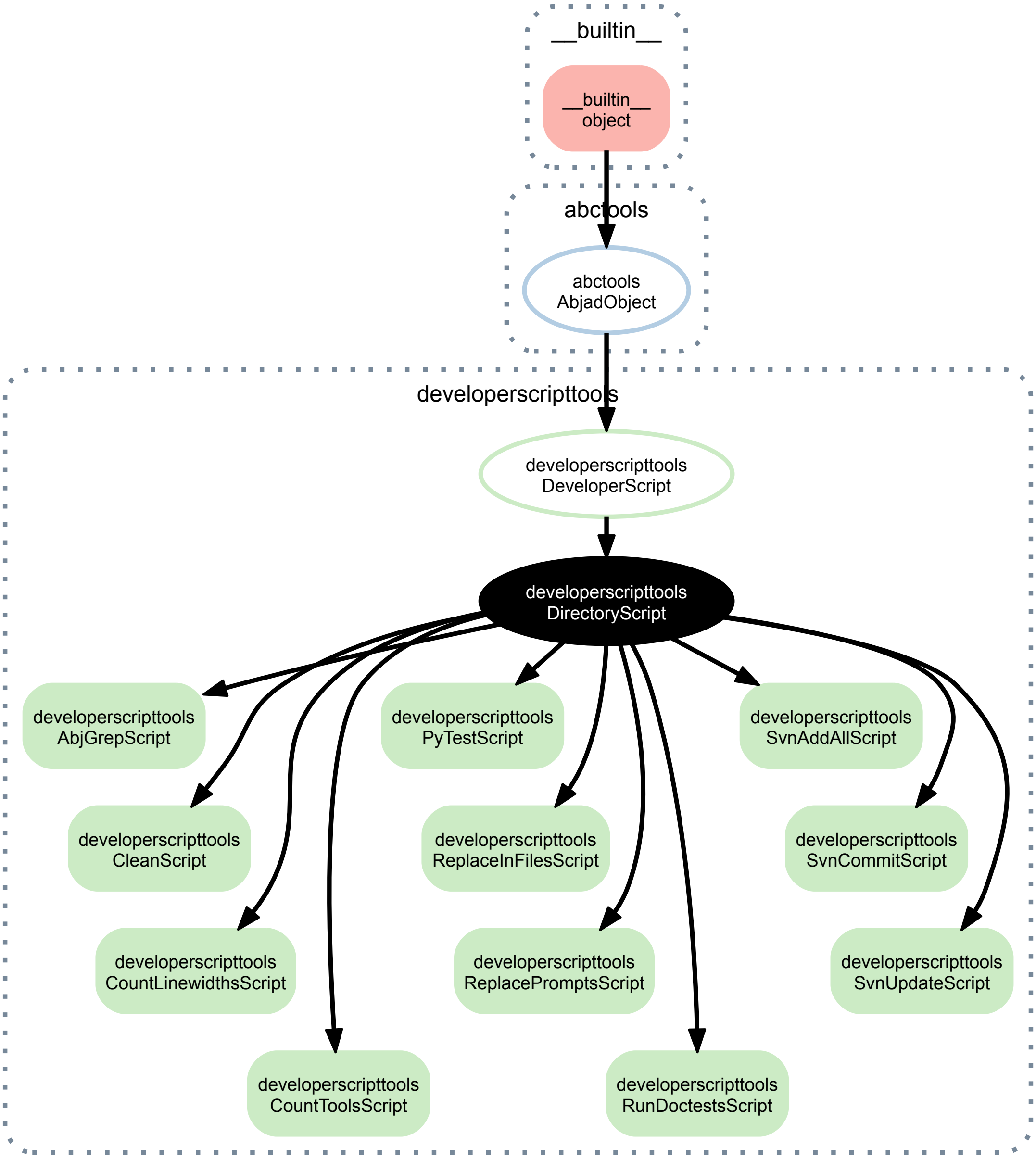
DeveloperScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

DeveloperScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.1.2 developerscripttools.DirectoryScript



class developerscripttools.**DirectoryScript**
DirectoryScript provides utilities for validating file system paths.

DirectoryScript is abstract.

Read-only properties

DirectoryScript.**alias**
The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

DirectoryScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

DirectoryScript.**formatted_help**

DirectoryScript.**formatted_usage**

DirectoryScript.**formatted_version**

DirectoryScript.**long_description**
The long description, printed after arguments explanations.

DirectoryScript.**program_name**
The name of the script, callable from the command line.

DirectoryScript.**scripting_group**
The script's scripting subcommand group.

DirectoryScript.**short_description**
The short description of the script, printed before arguments explanations.

Also used as a summary in other contexts.

DirectoryScript.**storage_format**
Storage format of Abjad object.

Return string.

DirectoryScript.**version**
The version number of the script.

Methods

DirectoryScript.**process_args**(*args*)

DirectoryScript.**setup_argument_parser**()

Special methods

DirectoryScript.**__call__**(*args=None*)

DirectoryScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

DirectoryScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectoryScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

DirectoryScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectoryScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

DirectoryScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

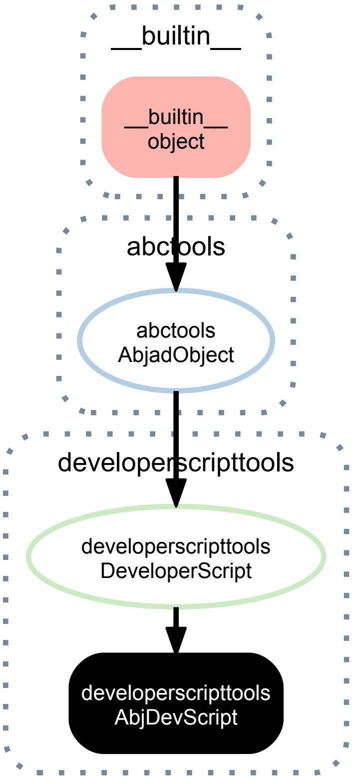
DirectoryScript.**__repr__**()

Interpreter representation of Abjad object.

Return string.

56.2 Concrete Classes

56.2.1 developerscripttools.AbjDevScript



class developerscripttools.**AbjDevScript**

AbjDevScript is the commandline entry-point to the Abjad developer scripts catalog.

Can be accessed on the commandline via *abj-dev* or *ajv*:

```
bash$ abj-dev
usage: abj-dev [-h] [--version]

               {help,list,api,book,clean,count,doctest,grep,new,rename,replace,svn}
               ...

Entry-point to Abjad developer scripts catalog.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit

subcommands:
  {help,list,api,book,clean,count,doctest,grep,new,rename,replace,svn}
    help                print subcommand help
    list                list subcommands
    api                 Build the Abjad APIs.
    book                Preprocess HTML, LaTeX or ReST source with Abjad.
    clean               Clean .pyc, __pycache__ and tmp* files and folders
                        from PATH.
```

```
        count                   "count"-related subcommands
        doctest                 Run doctests on all modules in current path.
        grep                    grep PATTERN in PATH
        new                     "new"-related subcommands
        rename                  Rename public modules.
        replace                 "replace"-related subcommands
        svn                     "svn"-related subcommands
```

ajv supports subcommands similar to *svn*:

```
bash$ ajv count -h
usage: abj-dev count [-h] {linewidths,tools} ...

optional arguments:
  -h, --help            show this help message and exit

count subcommands:
  {linewidths,tools}
    linewidths          Count maximum line-width of all modules in PATH.
    tools               Count tools in PATH.
```

Return *AbjDevScript* instance.

Read-only properties

AbjDevScript.**alias**
The alias to use for the script, useful only if the script defines an abj-dev scripting group as well.

AbjDevScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

AbjDevScript.**developer_script_aliases**

AbjDevScript.**developer_script_classes**

AbjDevScript.**developer_script_program_names**

AbjDevScript.**formatted_help**

AbjDevScript.**formatted_usage**

AbjDevScript.**formatted_version**

AbjDevScript.**long_description**

AbjDevScript.**program_name**
The name of the script, callable from the command line.

AbjDevScript.**scripting_group**
The script's scripting subcommand group.

AbjDevScript.**short_description**

AbjDevScript.**storage_format**
Storage format of Abjad object.

Return string.

AbjDevScript.**version**

Methods

AbjDevScript.**process_args**(*args*)

AbjDevScript.**setup_argument_parser**(*parser*)

Special methods

AbjDevScript.**__call__**(*args=None*)

AbjDevScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

AbjDevScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjDevScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjDevScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjDevScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

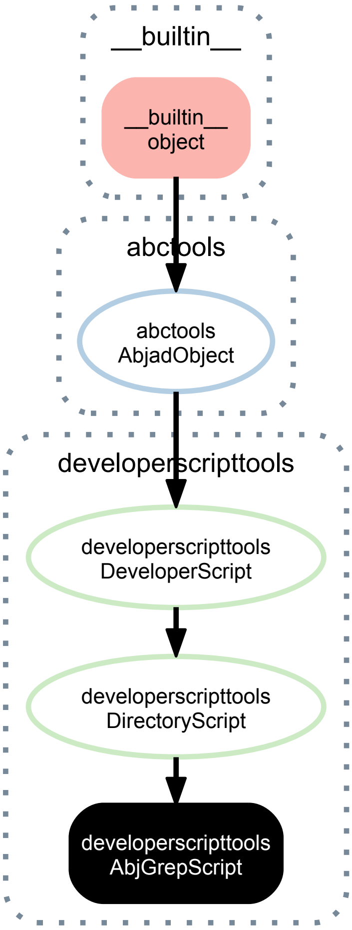
AbjDevScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjDevScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.2 developerscripttools.AbjGrepScript



class developerscripttools.**AbjGrepScript**

Run *grep* against a path, ignoring *svn* and docs-related files:

```
bash$ ajv grep -h
usage: abj-grep [-h] [--version] [-W] [-P PATH | -X | -M | -T | -R] pattern

grep PATTERN in PATH

positional arguments:
  pattern               pattern to search for

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -W, --whole-words-only
                        match only whole words, similar to grep's "-w" flag
  -P PATH, --path PATH  grep PATH
  -X, --abjad.tools    grep Abjad abjad.tools directory
  -M, --mainline        grep Abjad mainline directory
  -T, --tools           grep Abjad mainline tools directory
  -R, --root            grep Abjad root directory

If no PATH flag is specified, the current directory will be searched.
```

Return *AbjGrepScript* instance.

Read-only properties

AbjGrepScript.**alias**

AbjGrepScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

AbjGrepScript.**formatted_help**

AbjGrepScript.**formatted_usage**

AbjGrepScript.**formatted_version**

AbjGrepScript.**long_description**

AbjGrepScript.**program_name**
The name of the script, callable from the command line.

AbjGrepScript.**scripting_group**

AbjGrepScript.**short_description**

AbjGrepScript.**storage_format**
Storage format of Abjad object.

Return string.

AbjGrepScript.**version**

Methods

AbjGrepScript.**process_args**(*args*)

AbjGrepScript.**setup_argument_parser**(*parser*)

Special methods

AbjGrepScript.**__call__**(*args=None*)

AbjGrepScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

AbjGrepScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjGrepScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

AbjGrepScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

AbjGrepScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

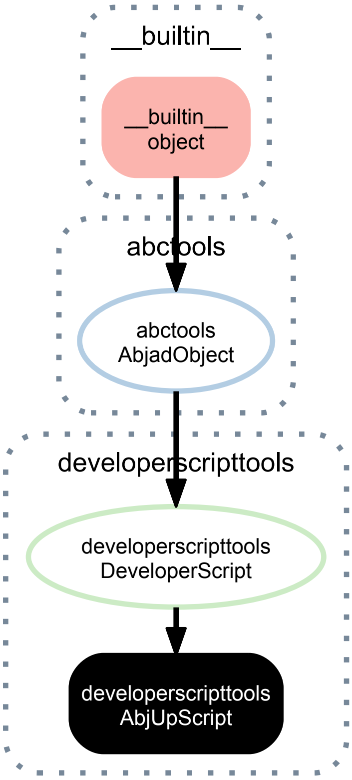
AbjGrepScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

AbjGrepScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.3 developerscripttools.AbjUpScript



class developerscripttools.**AbjUpScript**
Run *ajv svn up -R -C*:

```
bash$ ajv up -h
usage: abj-up [-h] [--version]

run `ajv svn up -R -C`

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

Return *AbjUpScript* instance.

Read-only properties

AbjUpScript.**alias**

AbjUpScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

AbjUpScript.**formatted_help**

AbjUpScript.**formatted_usage**

AbjUpScript.**formatted_version**

AbjUpScript.**long_description**

`AbjUpScript.`**`program_name`**
The name of the script, callable from the command line.

`AbjUpScript.`**`scripting_group`**

`AbjUpScript.`**`short_description`**

`AbjUpScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`AbjUpScript.`**`version`**

Methods

`AbjUpScript.`**`process_args`**`(`*args*`)`

`AbjUpScript.`**`setup_argument_parser`**`(`*parser*`)`

Special methods

`AbjUpScript.`**`__call__`**`(`*args=None*`)`

`AbjUpScript.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`AbjUpScript.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjUpScript.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`AbjUpScript.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjUpScript.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

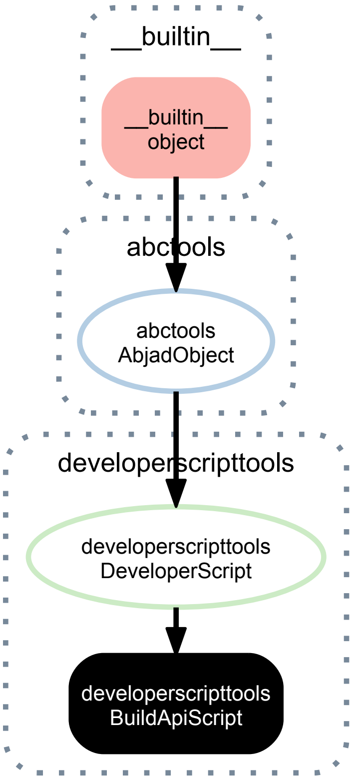
`AbjUpScript.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`AbjUpScript.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

56.2.4 developerscripttools.BuildApiScript



class developerscripttools.**BuildApiScript**

Build the Abjad APIs:

```
bash$ ajv api -h
usage: build-api [-h] [--version] [-M] [-X] [-C] [--format FORMAT]

Build the Abjad APIs.

optional arguments:
  -h, --help           show this help message and exit
  --version            show program's version number and exit
  -M, --mainline       build the mainline API
  -X, --abjad.tools  build the abjad.tools API
  -C, --clean          run "make clean" before building the api
  --format FORMAT      Sphinx builder to use
```

Return *BuildApiScript* instance.

Read-only properties

BuildApiScript.**alias**

BuildApiScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

BuildApiScript.**formatted_help**

BuildApiScript.**formatted_usage**

BuildApiScript.**formatted_version**

BuildApiScript.**long_description**

BuildApiScript.**program_name**

The name of the script, callable from the command line.

BuildApiScript.**scripting_group**

BuildApiScript.**short_description**

BuildApiScript.**storage_format**
Storage format of Abjad object.

Return string.

BuildApiScript.**version**

Methods

BuildApiScript.**process_args**(*args*)

BuildApiScript.**setup_argument_parser**(*parser*)

Special methods

BuildApiScript.**__call__**(*args=None*)

BuildApiScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

BuildApiScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BuildApiScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BuildApiScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BuildApiScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

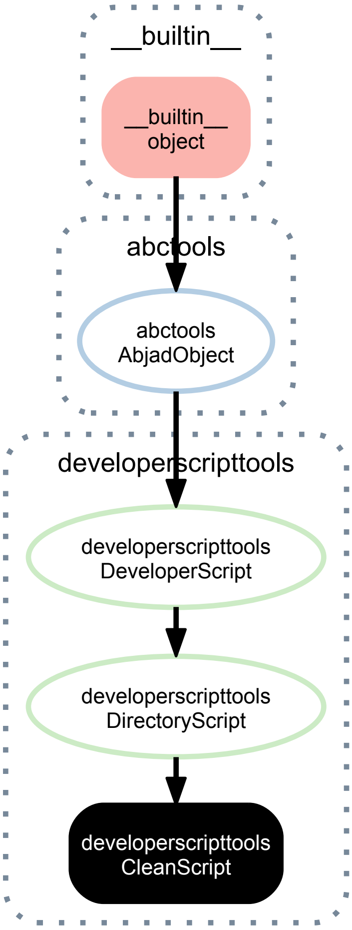
BuildApiScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

BuildApiScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.5 developerscripttools.CleanScript



class developerscripttools.**CleanScript**

Remove *.pyc, *.swp files and __pycache__ and tmp* directories recursively in a path:

```
bash$ ajv clean -h
usage: clean [-h] [--version] [--pyc] [--pycache] [--swp] [--tmp] path

Clean *.pyc, *.swp, __pycache__ and tmp* files and folders from PATH.

positional arguments:
  path        directory tree to be recursed over

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
  --pyc       delete *.pyc files
  --pycache   delete __pycache__ folders
  --swp       delete Vim *.swp file
  --tmp       delete tmp* folders
```

Return *CleanScript* instance.

Read-only properties

CleanScript.**alias**

CleanScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

`CleanScript.`**`formatted_help`**

`CleanScript.`**`formatted_usage`**

`CleanScript.`**`formatted_version`**

`CleanScript.`**`long_description`**

`CleanScript.`**`program_name`**
The name of the script, callable from the command line.

`CleanScript.`**`scripting_group`**

`CleanScript.`**`short_description`**

`CleanScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`CleanScript.`**`version`**

Methods

`CleanScript.`**`process_args`**(*args*)

`CleanScript.`**`setup_argument_parser`**(*parser*)

Special methods

`CleanScript.`**`__call__`**(*args=None*)

`CleanScript.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`CleanScript.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`CleanScript.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`CleanScript.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`CleanScript.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

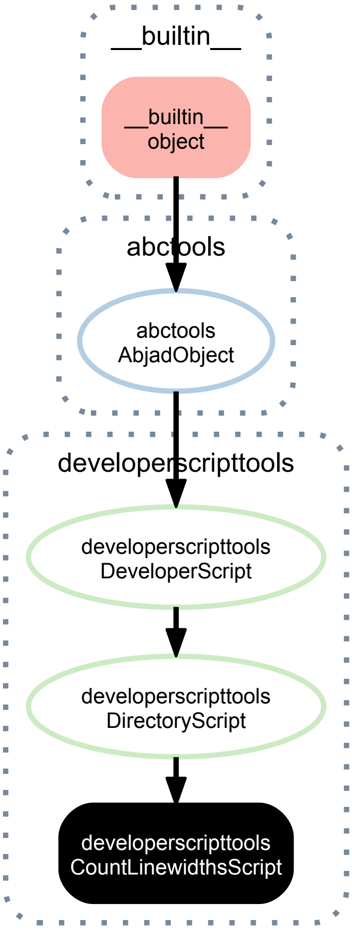
`CleanScript.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`CleanScript.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

56.2.6 developerscripttools.CountLinewidthsScript



class developerscripttools.**CountLinewidthsScript**

Tabulate the linewidths of modules in a path:

```
bash$ ajv count linewidths -h
usage: count-linewidths [-h] [--version] [-l N] [-o w|m] [-C | -D] [-a | -d]
                        [-gt N | -lt N | -eq N]
                        path

Count maximum line-width of all modules in PATH.

positional arguments:
  path                  directory tree to be recursed over

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -l N, --limit N       limit output to last N items
  -o w|m, --order-by w|m
                        order by line width [w] or module name [m]
  -C, --code            count linewidths of all code in module
  -D, --docstrings      count linewidths of all docstrings in module
  -a, --ascending       sort results ascending
  -d, --descending      sort results descending
  -gt N, --greater-than N
                        line widths greater than N
  -lt N, --less-than N  line widths less than N
  -eq N, --equal-to N   line widths equal to N
```

Return *CountLinewidthsScript* instance.

Read-only properties

CountLinewidthsScript.**alias**

CountLinewidthsScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

CountLinewidthsScript.**formatted_help**

CountLinewidthsScript.**formatted_usage**

CountLinewidthsScript.**formatted_version**

CountLinewidthsScript.**long_description**

CountLinewidthsScript.**program_name**
The name of the script, callable from the command line.

CountLinewidthsScript.**scripting_group**

CountLinewidthsScript.**short_description**

CountLinewidthsScript.**storage_format**
Storage format of Abjad object.

Return string.

CountLinewidthsScript.**version**

Methods

CountLinewidthsScript.**process_args**(*args*)

CountLinewidthsScript.**setup_argument_parser**(*parser*)

Special methods

CountLinewidthsScript.**__call__**(*args=None*)

CountLinewidthsScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

CountLinewidthsScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CountLinewidthsScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

CountLinewidthsScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

CountLinewidthsScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

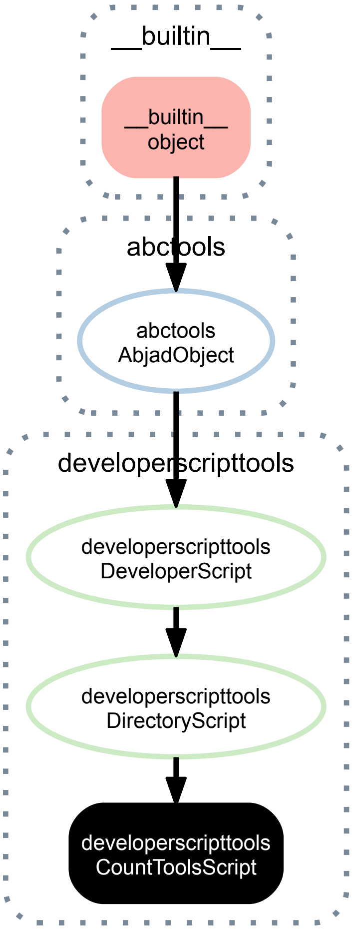
CountLinewidthsScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`CountLinewidthsScript.__repr__()`
Interpreter representation of Abjad object.

Return string.

56.2.7 developerscripttools.CountToolsScript



class `developerscripttools.CountToolsScript`
Count public and private functions and classes in a path:

```
bash$ ajv count tools -h
usage: count-tools [-h] [--version] path

Count tools in PATH.

positional arguments:
  path        directory tree to be recursed over

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

Return *CountToolsScript* instance.

Read-only properties

`CountToolsScript.alias`

`CountToolsScript.`**`argument_parser`**
The script's instance of argparse.ArgumentParser.

`CountToolsScript.`**`formatted_help`**

`CountToolsScript.`**`formatted_usage`**

`CountToolsScript.`**`formatted_version`**

`CountToolsScript.`**`long_description`**

`CountToolsScript.`**`program_name`**
The name of the script, callable from the command line.

`CountToolsScript.`**`scripting_group`**

`CountToolsScript.`**`short_description`**

`CountToolsScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`CountToolsScript.`**`version`**

Methods

`CountToolsScript.`**`process_args`**(*args*)

`CountToolsScript.`**`setup_argument_parser`**(*parser*)

Special methods

`CountToolsScript.`**`__call__`**(*args=None*)

`CountToolsScript.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`CountToolsScript.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`CountToolsScript.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`CountToolsScript.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`CountToolsScript.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

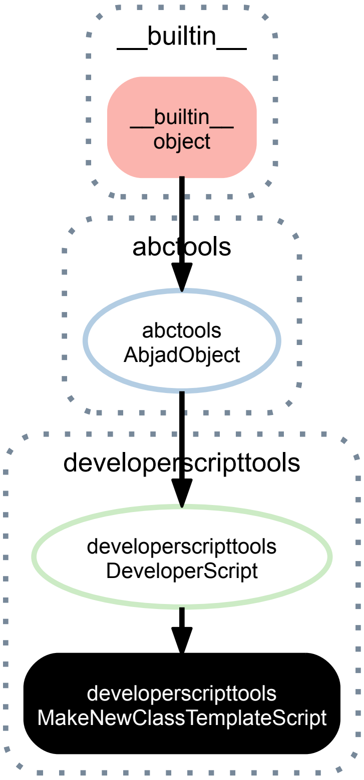
`CountToolsScript.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`CountToolsScript.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

56.2.8 developerscripttools.MakeNewClassTemplateScript



class developerscripttools.**MakeNewClassTemplateScript**

Create class stubs, complete with test subdirectory:

```
bash$ ajv new class -h
usage: make-new-class-template [-h] [--version] (-X | -M) name

Make a new class template file.

positional arguments:
  name                  tools package qualified class name

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -X, --experimental    use the Abjad experimental tools path
  -M, --mainline        use the Abjad mainline tools path
```

Return *MakeNewClassTemplateScript* instance.

Read-only properties

MakeNewClassTemplateScript.**alias**

MakeNewClassTemplateScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

MakeNewClassTemplateScript.**formatted_help**

MakeNewClassTemplateScript.**formatted_usage**

MakeNewClassTemplateScript.**formatted_version**

MakeNewClassTemplateScript.**long_description**

MakeNewClassTemplateScript.**program_name**
The name of the script, callable from the command line.

MakeNewClassTemplateScript.**scripting_group**

MakeNewClassTemplateScript.**short_description**

MakeNewClassTemplateScript.**storage_format**
Storage format of Abjad object.

Return string.

MakeNewClassTemplateScript.**version**

Methods

MakeNewClassTemplateScript.**process_args**(*args*)

MakeNewClassTemplateScript.**setup_argument_parser**(*parser*)

Special methods

MakeNewClassTemplateScript.**__call__**(*args=None*)

MakeNewClassTemplateScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

MakeNewClassTemplateScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MakeNewClassTemplateScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

MakeNewClassTemplateScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

MakeNewClassTemplateScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

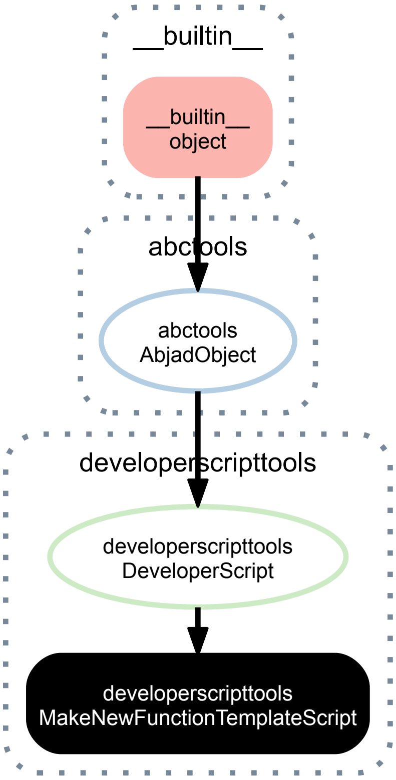
MakeNewClassTemplateScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

MakeNewClassTemplateScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.9 developerscripttools.MakeNewFunctionTemplateScript



class developerscripttools.**MakeNewFunctionTemplateScript**

Create function stub files:

```
bash$ ajv new function -h
usage: make-new-function-template [-h] [--version] (-X | -M) name

Make a new function template file.

positional arguments:
  name                  tools package qualified function name

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -X, --experimental    use the Abjad experimental path
  -M, --mainline        use the Abjad mainline tools path
```

Return *MakeNewFunctionTemplateScript* instance.

Read-only properties

MakeNewFunctionTemplateScript.**alias**

MakeNewFunctionTemplateScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

MakeNewFunctionTemplateScript.**formatted_help**

MakeNewFunctionTemplateScript.**formatted_usage**

MakeNewFunctionTemplateScript.**formatted_version**

MakeNewFunctionTemplateScript.**long_description**

`MakeNewFunctionTemplateScript.`**`program_name`**
The name of the script, callable from the command line.

`MakeNewFunctionTemplateScript.`**`scripting_group`**

`MakeNewFunctionTemplateScript.`**`short_description`**

`MakeNewFunctionTemplateScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`MakeNewFunctionTemplateScript.`**`version`**

Methods

`MakeNewFunctionTemplateScript.`**`process_args`**(*args*)

`MakeNewFunctionTemplateScript.`**`setup_argument_parser`**(*parser*)

Special methods

`MakeNewFunctionTemplateScript.`**`__call__`**(*args=None*)

`MakeNewFunctionTemplateScript.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`MakeNewFunctionTemplateScript.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MakeNewFunctionTemplateScript.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`MakeNewFunctionTemplateScript.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`MakeNewFunctionTemplateScript.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

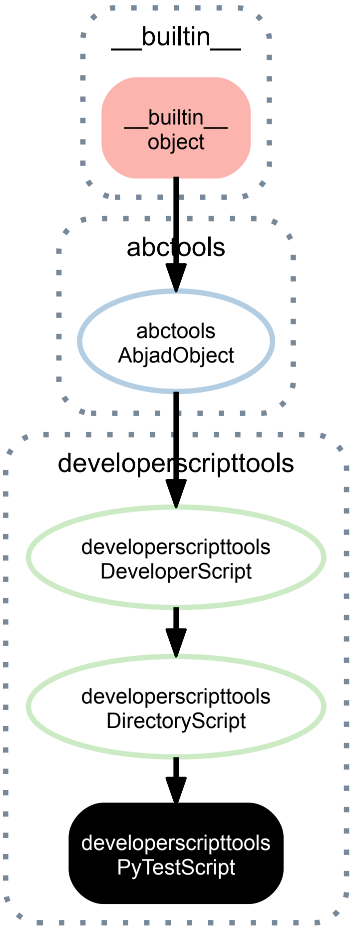
`MakeNewFunctionTemplateScript.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`MakeNewFunctionTemplateScript.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

56.2.10 developerscripttools.PyTestScript



class `developerscripttools.`**`PyTestScript`**

Run *py.test* on various Abjad paths:

```
bash$ ajv test -h
usage: py-test [-h] [--version] [-p] [-x] [-A | -D | -M | -X]

Run "py.test" on various Abjad paths.

optional arguments:
  -h, --help          show this help message and exit
  --version           show program's version number and exit
  -p, --parallel      run py.test with multiprocessing
  -x, --exitfirst     stop on first failure
  -A, --all           test all directories, including demos
  -D, --demos         test demos directory
  -M, --mainline      test mainline tools directory
  -X, --experimental  test experimental directory
```

Return *PyTestScript* instance.

Read-only properties

`PyTestScript.`**`alias`**

`PyTestScript.`**`argument_parser`**

The script's instance of argparse.ArgumentParser.

`PyTestScript.`**`formatted_help`**

PyTestScript.**formatted_usage**

PyTestScript.**formatted_version**

PyTestScript.**long_description**

PyTestScript.**program_name**
The name of the script, callable from the command line.

PyTestScript.**scripting_group**

PyTestScript.**short_description**

PyTestScript.**storage_format**
Storage format of Abjad object.

Return string.

PyTestScript.**version**

Methods

PyTestScript.**process_args**(*args*)

PyTestScript.**setup_argument_parser**(*parser*)

Special methods

PyTestScript.**__call__**(*args=None*)

PyTestScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

PyTestScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PyTestScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

PyTestScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

PyTestScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

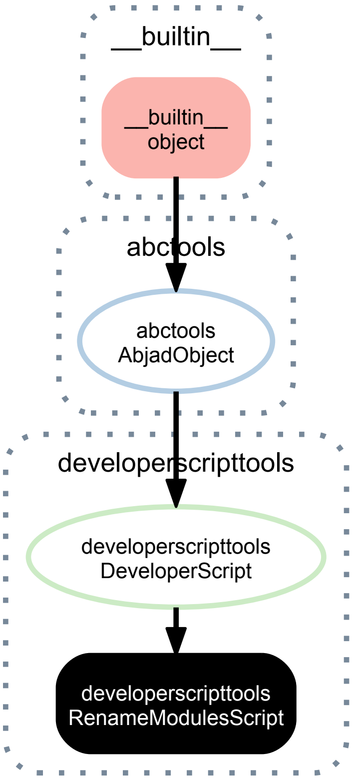
PyTestScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

PyTestScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.11 developerscripttools.RenameModulesScript



class developerscripttools.**RenameModulesScript**

Rename classes and functions.

Handle renaming the module and package, as well as any tests, documentation or mentions of the class throughout the Abjad codebase:

```
$ ajv rename -h
usage: rename-modules [-h] [--version] (-C | -F)

Rename public modules.

optional arguments:
  -h, --help       show this help message and exit
  --version        show program's version number and exit
  -C, --classes    rename classes
  -F, --functions  rename functions
```

Return *RenameModulesScript* instance.

Read-only properties

RenameModulesScript.**alias**

RenameModulesScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

RenameModulesScript.**formatted_help**

RenameModulesScript.**formatted_usage**

RenameModulesScript.**formatted_version**

RenameModulesScript.**long_description**

RenameModulesScript.**program_name**
The name of the script, callable from the command line.

RenameModulesScript.**scripting_group**

RenameModulesScript.**short_description**

RenameModulesScript.**storage_format**
Storage format of Abjad object.

Return string.

RenameModulesScript.**version**

Methods

RenameModulesScript.**process_args**(*args*)

RenameModulesScript.**setup_argument_parser**(*parser*)

Special methods

RenameModulesScript.**__call__**(*args=None*)

RenameModulesScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

RenameModulesScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RenameModulesScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

RenameModulesScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RenameModulesScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

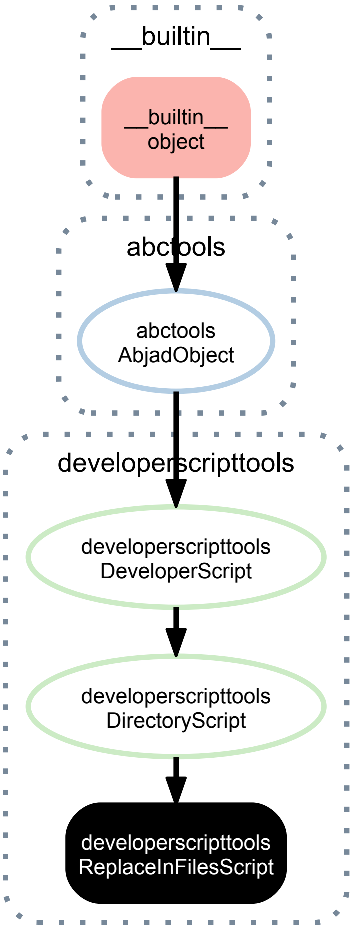
RenameModulesScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

RenameModulesScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.12 developerscripttools.ReplaceInFilesScript



class `developerscripttools.`**`ReplaceInFilesScript`**

Replace text in files recursively:

```
bash$ ajv replace text -h
usage: replace-in-files [-h] [--version] [--verbose] [-Y] [-R] [-W]
                        [-F PATTERN] [-D PATTERN]
                        path old new

Replace text.

positional arguments:
  path                  directory tree to be recursed over
  old                   old text
  new                   new text

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --verbose             print replacement info even when --force flag is set.
  -Y, --force           force "yes" to every replacement
  -R, --regex           treat "old" as a regular expression
  -W, --whole-words-only
                        match only whole words, similar to grep's "-w" flag
  -F PATTERN, --without-files PATTERN
                        Exclude files matching pattern(s)
  -D PATTERN, --without-dirs PATTERN
                        Exclude folders matching pattern(s)
```

Multiple patterns for excluding files or folders can be specified by restating the *–without-files* or *–without-*

dirs commands:

```
bash$ ajv replace text . foo bar -F *.txt -F *.rst -F *.htm
```

Return *ReplaceInFilesScript* instance.

Read-only properties

ReplaceInFilesScript.**alias**

ReplaceInFilesScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

ReplaceInFilesScript.**formatted_help**

ReplaceInFilesScript.**formatted_usage**

ReplaceInFilesScript.**formatted_version**

ReplaceInFilesScript.**long_description**

ReplaceInFilesScript.**program_name**
The name of the script, callable from the command line.

ReplaceInFilesScript.**scripting_group**

ReplaceInFilesScript.**short_description**

ReplaceInFilesScript.**skipped_directories**

ReplaceInFilesScript.**skipped_files**

ReplaceInFilesScript.**storage_format**
Storage format of Abjad object.

Return string.

ReplaceInFilesScript.**version**

Methods

ReplaceInFilesScript.**process_args**(*args*)

ReplaceInFilesScript.**setup_argument_parser**(*parser*)

Special methods

ReplaceInFilesScript.**__call__**(*args=None*)

ReplaceInFilesScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ReplaceInFilesScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReplaceInFilesScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReplaceInFilesScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReplaceInFilesScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

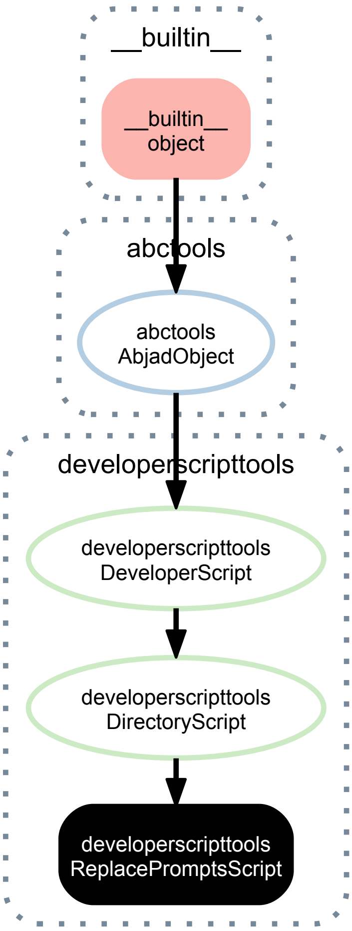
ReplaceInFilesScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ReplaceInFilesScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.13 developerscripttools.ReplacePromptsScript



class developerscripttools.**ReplacePromptsScript**
Replace prompts in code examples recursively:

```
bash$ ajv replace prompts -h
usage: replace-prompts [-h] [--version] [-PA | -AP] path

Replace prompts.

positional arguments:
  path                  directory tree to be recursed over

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
```

```
    -PA, --python-to-abjad
                          replace Python (">>>") prompts with Abjad ("abjad>")
                          prompts
    -AP, --abjad-to-python
                          replace Abjad ("abjad>") prompts with Python (">>>")
                          prompts

  examples:

    Replace Python prompts with Abjad prompts in the current directory:

    $ abj-dev replace prompts --python-to-abjad .

    Replace Abjad prompts with Python prompts in the grandparent directory:

    $ abj-dev replace prompts --abjad-to-python ../..
```

ReplacePromptsScript uses *ReplaceInFilesScript* for its replacement functionality.

Return *ReplacePromptsScript* instance.

Read-only properties

ReplacePromptsScript.**alias**

ReplacePromptsScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

ReplacePromptsScript.**formatted_help**

ReplacePromptsScript.**formatted_usage**

ReplacePromptsScript.**formatted_version**

ReplacePromptsScript.**long_description**

ReplacePromptsScript.**program_name**
The name of the script, callable from the command line.

ReplacePromptsScript.**scripting_group**

ReplacePromptsScript.**short_description**

ReplacePromptsScript.**storage_format**
Storage format of Abjad object.

Return string.

ReplacePromptsScript.**version**

Methods

ReplacePromptsScript.**process_args**(*args*)

ReplacePromptsScript.**setup_argument_parser**(*parser*)

Special methods

ReplacePromptsScript.**__call__**(*args=None*)

ReplacePromptsScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ReplacePromptsScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReplacePromptsScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReplacePromptsScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReplacePromptsScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

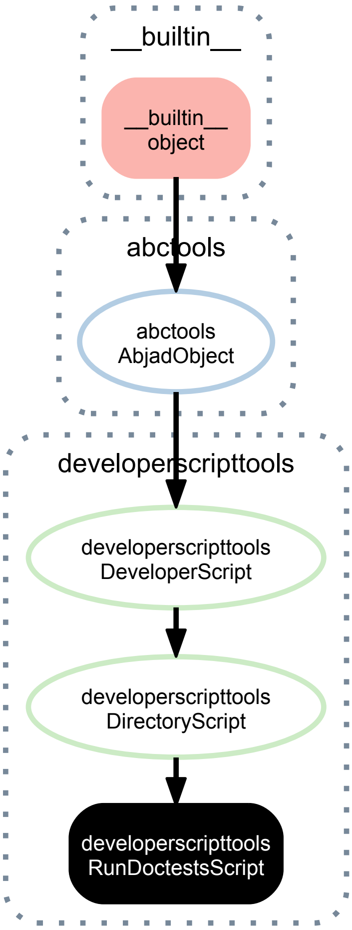
ReplacePromptsScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ReplacePromptsScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.14 developerscripttools.RunDoctestsScript



class developerscripttools.**RunDoctestsScript**

Run doctests on all Python files in current directory recursively:

```
bash$ ajv doctest -h
usage: run-doctests [-h] [--version]

Run doctests on all modules in current path.

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

Return *RunDoctestsScript* instance.

Read-only properties

RunDoctestsScript.**alias**

RunDoctestsScript.**argument_parser**

The script's instance of argparse.ArgumentParser.

RunDoctestsScript.**formatted_help**

RunDoctestsScript.**formatted_usage**

RunDoctestsScript.**formatted_version**

RunDoctestsScript.**long_description**

RunDoctestsScript.**program_name**

The name of the script, callable from the command line.

RunDoctestsScript.**scripting_group**

RunDoctestsScript.**short_description**

RunDoctestsScript.**storage_format**

Storage format of Abjad object.

Return string.

RunDoctestsScript.**version**

Methods

RunDoctestsScript.**process_args**(*args*)

RunDoctestsScript.**setup_argument_parser**(*parser*)

Special methods

RunDoctestsScript.**__call__**(*args=None*)

RunDoctestsScript.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

RunDoctestsScript.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

RunDoctestsScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

RunDoctestsScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

RunDoctestsScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

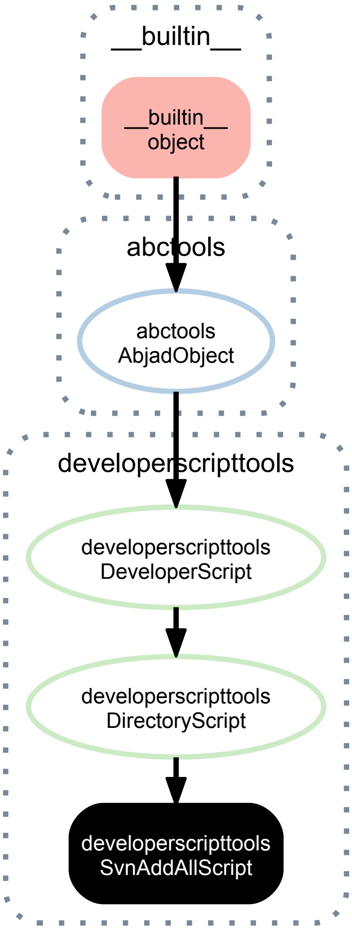
RunDoctestsScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

RunDoctestsScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.15 developerscripttools.SvnAddAllScript



class developerscripttools.**SvnAddAllScript**
Run *svn add* on all unversioned files in path:

```
bash$ ajv svn add -h
usage: svn-add-all [-h] [--version] path
```

```
"svn add" all unversioned files in PATH.

positional arguments:
  path        directory tree to be recursed over

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

Return *SvnAddAllScript* instance.

Read-only properties

SvnAddAllScript.**alias**

SvnAddAllScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

SvnAddAllScript.**formatted_help**

SvnAddAllScript.**formatted_usage**

SvnAddAllScript.**formatted_version**

SvnAddAllScript.**long_description**

SvnAddAllScript.**program_name**
The name of the script, callable from the command line.

SvnAddAllScript.**scripting_group**

SvnAddAllScript.**short_description**

SvnAddAllScript.**storage_format**
Storage format of Abjad object.

Return string.

SvnAddAllScript.**version**

Methods

SvnAddAllScript.**process_args**(*args*)

SvnAddAllScript.**setup_argument_parser**(*parser*)

Special methods

SvnAddAllScript.**__call__**(*args=None*)

SvnAddAllScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

SvnAddAllScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnAddAllScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SvnAddAllScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnAddAllScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

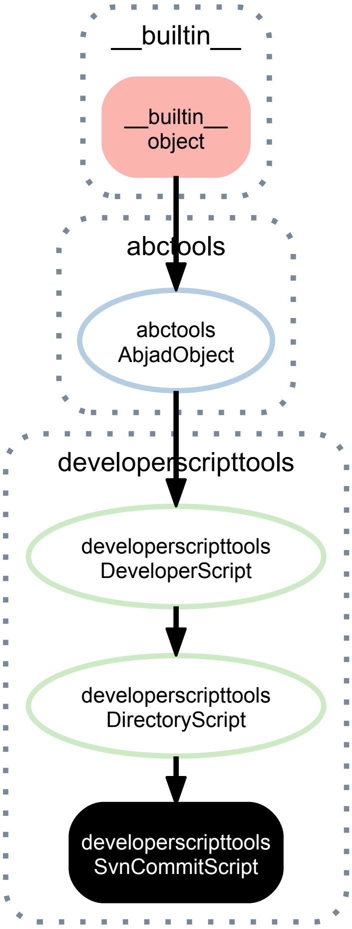
SvnAddAllScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SvnAddAllScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.16 developerscripttools.SvnCommitScript



class developerscripttools.**SvnCommitScript**
Run *svn commit*, using the commit message stored in the *.abjad* directory.

The commit message will be printed to the terminal, and must be manually accepted or rejected before proceeding:

```
bash$ ajv svn ci -h
usage: svn-commit [-h] [--version] path

"svn commit", using previously written commit message.
```

```
positional arguments:
  path        commit the path PATH

optional arguments:
  -h, --help  show this help message and exit
  --version   show program's version number and exit
```

Return *SvnCommitScript* instance.

Read-only properties

`SvnCommitScript.`**`alias`**

`SvnCommitScript.`**`argument_parser`**
The script's instance of argparse.ArgumentParser.

`SvnCommitScript.`**`formatted_help`**

`SvnCommitScript.`**`formatted_usage`**

`SvnCommitScript.`**`formatted_version`**

`SvnCommitScript.`**`long_description`**

`SvnCommitScript.`**`program_name`**
The name of the script, callable from the command line.

`SvnCommitScript.`**`scripting_group`**

`SvnCommitScript.`**`short_description`**

`SvnCommitScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`SvnCommitScript.`**`version`**

Methods

`SvnCommitScript.`**`process_args`**(*args*)

`SvnCommitScript.`**`setup_argument_parser`**(*parser*)

Special methods

`SvnCommitScript.`**`__call__`**(*args=None*)

`SvnCommitScript.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`SvnCommitScript.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`SvnCommitScript.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SvnCommitScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnCommitScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

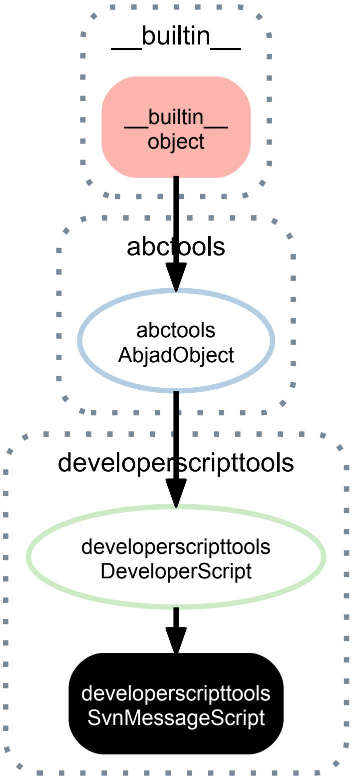
SvnCommitScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SvnCommitScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.17 developerscripttools.SvnMessageScript



class developerscripttools.**SvnMessageScript**
Edit a temporary *svn* commit message, stored in the *.abjad* directory:

```
bash$ ajv svn msg -h
usage: svn-message [-h] [--version] [-C]

Write commit message for future commit usage.

optional arguments:
  -h, --help    show this help message and exit
  --version     show program's version number and exit
  -C, --clean   delete previous commit message before editing
```

Return *SvnMessageScript* instance.

Read-only properties

SvnMessageScript.**alias**

SvnMessageScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

SvnMessageScript.**commit_message_path**

SvnMessageScript.**formatted_help**

SvnMessageScript.**formatted_usage**

SvnMessageScript.**formatted_version**

SvnMessageScript.**long_description**

SvnMessageScript.**program_name**
The name of the script, callable from the command line.

SvnMessageScript.**scripting_group**

SvnMessageScript.**short_description**

SvnMessageScript.**storage_format**
Storage format of Abjad object.

Return string.

SvnMessageScript.**version**

Methods

SvnMessageScript.**process_args**(*args*)

SvnMessageScript.**setup_argument_parser**(*parser*)

Special methods

SvnMessageScript.**__call__**(*args=None*)

SvnMessageScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

SvnMessageScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnMessageScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SvnMessageScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnMessageScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

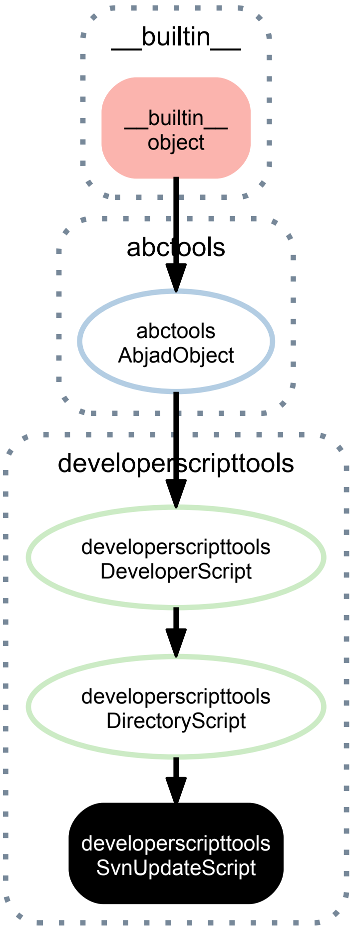
`SvnMessageScript.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`SvnMessageScript.__repr__()`
Interpreter representation of Abjad object.

Return string.

56.2.18 developerscripttools.SvnUpdateScript



class `developerscripttools.SvnUpdateScript`
Run *svn up* on various Abjad paths:

```
bash$ ajv svn up -h
usage: svn-update [-h] [--version] [-C] [-P PATH | -E | -M | -R]

"svn update" various paths.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -C, --clean           remove .pyc files and __pycache__ directories before
                        updating
  -P PATH, --path PATH  update the path PATH
  -E, --abjad.tools    update Abjad abjad.tools directory
  -M, --mainline        update Abjad mainline directory
  -R, --root            update Abjad root directory
```

```
If no path flag is specified, the current directory will be updated.
```

It is usually most useful to run the script with the *–clean* flag, in case there are incoming deletes, as *svn* will not delete directories containing unversioned files, such as .pyc:

```
bash$ ajv svn up -C -R
```

Return *SvnUpdateScript* instance.

Read-only properties

SvnUpdateScript.**alias**

SvnUpdateScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

SvnUpdateScript.**formatted_help**

SvnUpdateScript.**formatted_usage**

SvnUpdateScript.**formatted_version**

SvnUpdateScript.**long_description**

SvnUpdateScript.**program_name**
The name of the script, callable from the command line.

SvnUpdateScript.**scripting_group**

SvnUpdateScript.**short_description**

SvnUpdateScript.**storage_format**
Storage format of Abjad object.

Return string.

SvnUpdateScript.**version**

Methods

SvnUpdateScript.**process_args**(*args*)

SvnUpdateScript.**setup_argument_parser**(*parser*)

Special methods

SvnUpdateScript.**__call__**(*args=None*)

SvnUpdateScript.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

SvnUpdateScript.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnUpdateScript.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SvnUpdateScript.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SvnUpdateScript.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

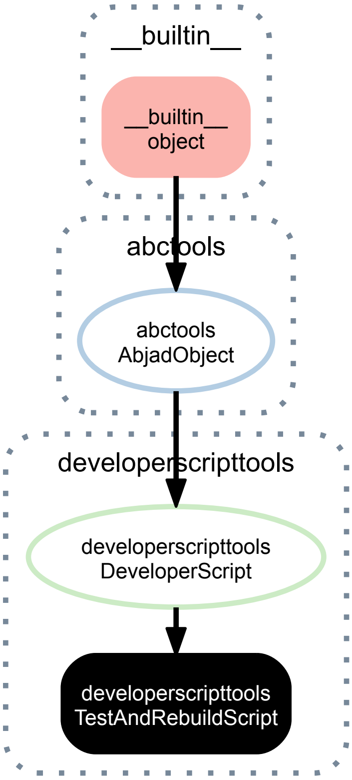
SvnUpdateScript.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SvnUpdateScript.**__repr__**()
Interpreter representation of Abjad object.

Return string.

56.2.19 developerscripttools.TestAndRebuildScript



class developerscripttools.**TestAndRebuildScript**

Read-only properties

TestAndRebuildScript.**alias**

TestAndRebuildScript.**argument_parser**
The script's instance of argparse.ArgumentParser.

TestAndRebuildScript.**formatted_help**

TestAndRebuildScript.**formatted_usage**

TestAndRebuildScript.**formatted_version**

`TestAndRebuildScript.`**`long_description`**

`TestAndRebuildScript.`**`program_name`**
The name of the script, callable from the command line.

`TestAndRebuildScript.`**`scripting_group`**

`TestAndRebuildScript.`**`short_description`**

`TestAndRebuildScript.`**`storage_format`**
Storage format of Abjad object.

Return string.

`TestAndRebuildScript.`**`version`**

Methods

`TestAndRebuildScript.`**`get_terminal_width`**`()`
Borrowed from the py lib.

`TestAndRebuildScript.`**`process_args`**`(`*args*`)`

`TestAndRebuildScript.`**`rebuild_docs`**`(`*args*`)`

`TestAndRebuildScript.`**`run_doctest`**`(`*args*`)`

`TestAndRebuildScript.`**`run_pytest`**`(`*args*`)`

`TestAndRebuildScript.`**`setup_argument_parser`**`(`*parser*`)`

Special methods

`TestAndRebuildScript.`**`__call__`**`(`*args=None*`)`

`TestAndRebuildScript.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`TestAndRebuildScript.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TestAndRebuildScript.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`TestAndRebuildScript.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TestAndRebuildScript.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`TestAndRebuildScript.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`TestAndRebuildScript.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

56.3 Functions

56.3.1 developerscripttools.get_developer_script_classes

developerscripttools.**get_developer_script_classes**()

Return a list of all developer script classes.

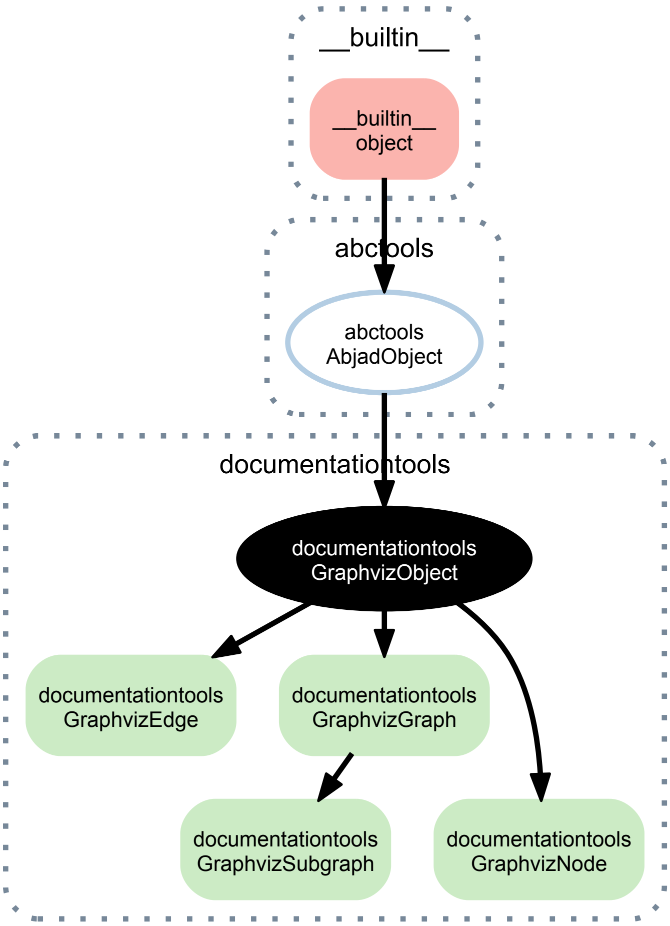
CHAPTER

FIFTYSEVEN

DOCUMENTATIONTOOLS

57.1 Abstract Classes

57.1.1 documentationtools.GraphvizObject



class `documentationtools.`**`GraphvizObject`**(*attributes=None*)

An attributed Graphviz object.

Read-only properties

`GraphvizObject.`**`attributes`**

`GraphvizObject.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

GraphvizObject.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

GraphvizObject.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizObject.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GraphvizObject.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizObject.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

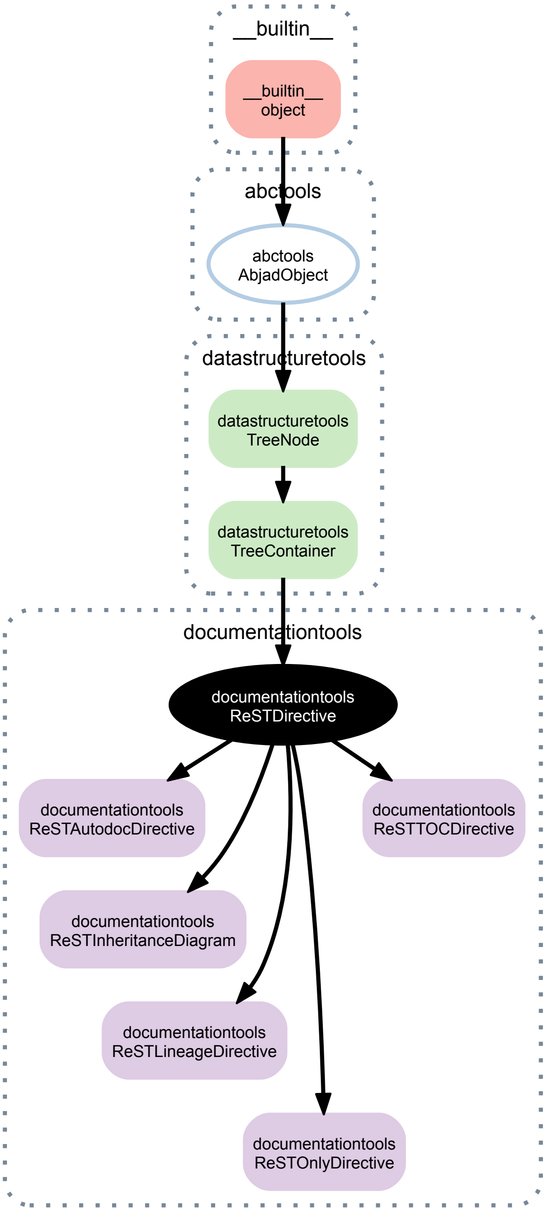
GraphvizObject.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

GraphvizObject.**__repr__**()
Interpreter representation of Abjad object.

Return string.

57.1.2 documentationtools.ReSTDirective



class `documentationtools.`**`ReSTDirective`**(*argument=None*, *children=None*, *name=None*, *options=None*)

Read-only properties

`ReSTDirective.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`ReSTDirective.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`ReSTDirective.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

`ReSTDirective.`**`directive`**

`ReSTDirective.`**`graph_order`**

ReSTDirective.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTDirective.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

ReSTDirective.**node_klass**

ReSTDirective.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`ReSTDirective.options`

`ReSTDirective.parent`

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`ReSTDirective.proper_parentage`

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

`ReSTDirective.rest_format`

`ReSTDirective.root`

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
```

```
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTDirective.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

ReSTDirective.**argument**

ReSTDirective.**name**

Methods

ReSTDirective.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTDirective.**extend**(*expr*)
Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTDirective.**index**(*node*)
Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

ReSTDirective.**insert**(*i*, *node*)
Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTDirective.**pop**(*i=-1*)
Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTDirective.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTDirective.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTDirective.**__copy__**(**args*)

ReSTDirective.**__deepcopy__**(**args*)

ReSTDirective.**__delitem__**(*i*)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

`ReSTDirective.__eq__(expr)`
True if type, duration and children are equivalent, otherwise False.

Return boolean.

`ReSTDirective.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ReSTDirective.__getitem__(i)`
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

`ReSTDirective.__getstate__()`

ReSTDirective.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTDirective.**__iter__**()

ReSTDirective.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTDirective.**__len__**()
Return nonnegative integer number of nodes in container.

ReSTDirective.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTDirective.**__ne__**(*expr*)

ReSTDirective.**__repr__**()

ReSTDirective.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

```
>>> b.parent is None
True
```

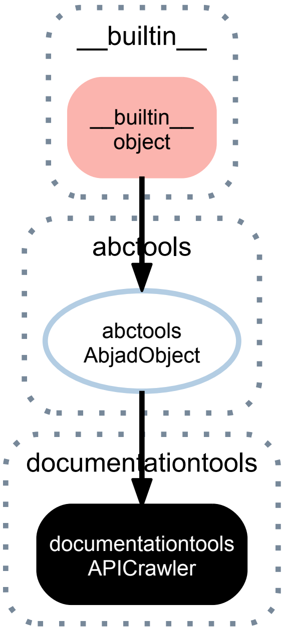
```
>>> a.children == (c,)
True
```

Return *None*.

ReSTDirective.**__setstate__**(*state*)

57.2 Concrete Classes

57.2.1 documentationtools.APICrawler



class documentationtools.**APICrawler**(*code_root, docs_root, root_package_name, ignored_directories=['test', '.svn', '__pycache__'], prefix='abjad.tools.'*)

Generates directories containing ReST to parallel directories containing code.

Read-only properties

APICrawler.**code_root**

APICrawler.**docs_root**

APICrawler.**module_crawler**

APICrawler.**prefix**

APICrawler.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

APICrawler.**__call__**()

Crawl *code_root* and generate corresponding ReST in *docs_root* while ignoring ignored directories.

APICrawler.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

APICrawler.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`APICrawler.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`APICrawler.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`APICrawler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

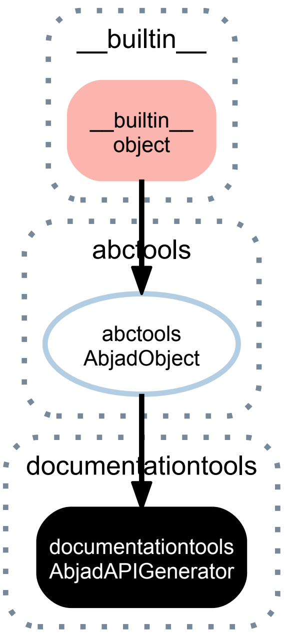
`APICrawler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`APICrawler.__repr__()`
Interpreter representation of Abjad object.

Return string.

57.2.2 documentationtools.AbjadAPIGenerator



class `documentationtools.AbjadAPIGenerator`
Creates Abjad's API ReST:

- writes ReST pages for individual classes and functions
- writes the API index ReST
- handles sorting tools packages into composition, manual-loading and unstable
- handles ignoring private tools packages

Returns *AbjadAPIGenerator* instance.

Read-only properties

`AbjadAPIGenerator.`**`docs_api_index_path`**
Path to index.rst for Abjad API.

`AbjadAPIGenerator.`**`package_prefix`**

`AbjadAPIGenerator.`**`path_definitions`**
Code path / docs path / package prefix triples.

`AbjadAPIGenerator.`**`root_package`**

`AbjadAPIGenerator.`**`storage_format`**
Storage format of Abjad object.

Return string.

`AbjadAPIGenerator.`**`tools_package_path_index`**

Special methods

`AbjadAPIGenerator.`**`__call__`**`(`*verbose=False*`)`

`AbjadAPIGenerator.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`AbjadAPIGenerator.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjadAPIGenerator.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`AbjadAPIGenerator.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`AbjadAPIGenerator.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

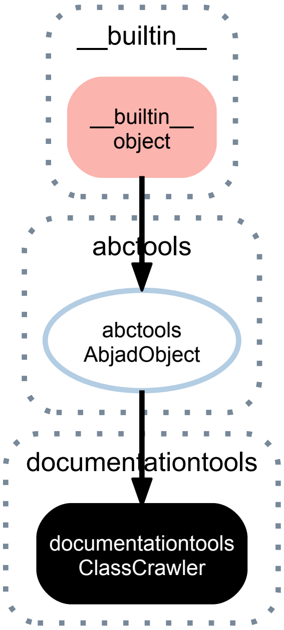
`AbjadAPIGenerator.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`AbjadAPIGenerator.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

57.2.3 documentationtools.ClassCrawler



class documentationtools.**ClassCrawler**(*code_root*, *include_private_objects=False*, *root_package_name=None*)

Read-only properties

ClassCrawler.**code_root**

ClassCrawler.**include_private_objects**

ClassCrawler.**module_crawler**

ClassCrawler.**root_package_name**

ClassCrawler.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

ClassCrawler.**__call__**()

ClassCrawler.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ClassCrawler.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ClassCrawler.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ClassCrawler.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ClassCrawler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

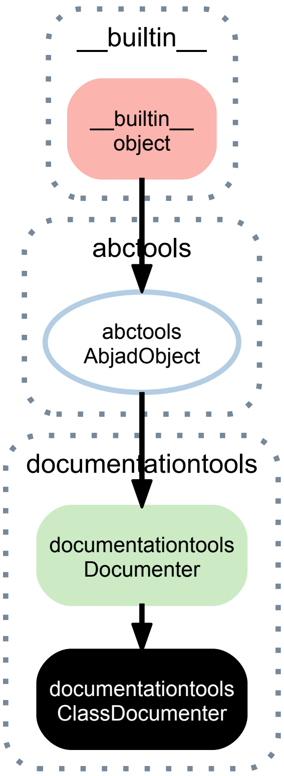
`ClassCrawler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`ClassCrawler.__repr__()`
Interpreter representation of Abjad object.

Return string.

57.2.4 documentationtools.ClassDocumenter



class `documentationtools.ClassDocumenter`(*obj*, *prefix='abjad.tools.'*)
ClassDocumenter generates an ReST API entry for a given class:

```
>>> documenter = documentationtools.ClassDocumenter(notetools.Note)
>>> rest = documenter()
```

Returns `ClassDocumenter` instance.

Read-only properties

`ClassDocumenter.data`

`ClassDocumenter.inherited_attributes`

`ClassDocumenter.is_abstract`

`ClassDocumenter.methods`

`ClassDocumenter.module_name`

ClassDocumenter.**object**

ClassDocumenter.**prefix**

ClassDocumenter.**readonly_properties**

ClassDocumenter.**readwrite_properties**

ClassDocumenter.**special_methods**

ClassDocumenter.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

ClassDocumenter.**__call__**()
Generate documentation.

Returns string.

ClassDocumenter.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ClassDocumenter.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ClassDocumenter.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ClassDocumenter.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ClassDocumenter.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

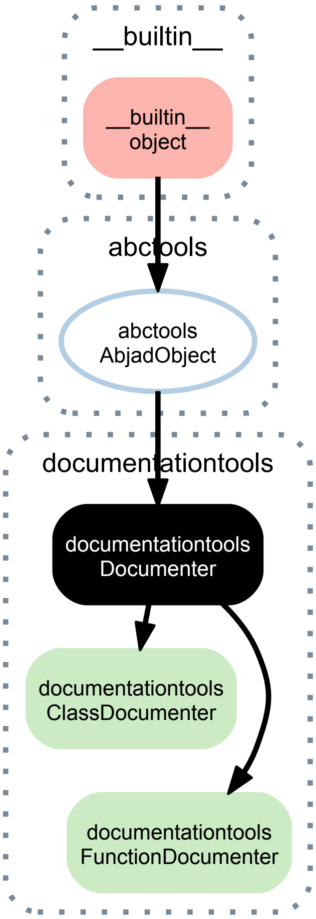
ClassDocumenter.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ClassDocumenter.**__repr__**()
Interpreter representation of Abjad object.

Return string.

57.2.5 documentationtools.Documenter



class `documentationtools.`**`Documenter`**(*obj*, *prefix='abjad.tools.'*)

Documenter is an abstract base class for documentation classes.

Read-only properties

`Documenter.`**`module_name`**

`Documenter.`**`object`**

`Documenter.`**`prefix`**

`Documenter.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`Documenter.`**`__call__`**()

`Documenter.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`Documenter.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

Documenter.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

Documenter.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

Documenter.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

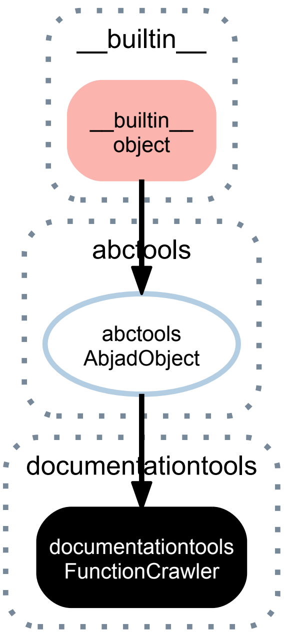
Documenter.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

Documenter.**__repr__**()
Interpreter representation of Abjad object.

Return string.

57.2.6 documentationtools.FunctionCrawler



class documentationtools.**FunctionCrawler**(*code_root*, *include_private_objects=False*, *root_package_name=None*)

Read-only properties

FunctionCrawler.**code_root**

FunctionCrawler.**include_private_objects**

FunctionCrawler.**module_crawler**

FunctionCrawler.**root_package_name**

FunctionCrawler.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

`FunctionCrawler.__call__()`

`FunctionCrawler.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`FunctionCrawler.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`FunctionCrawler.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`FunctionCrawler.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`FunctionCrawler.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

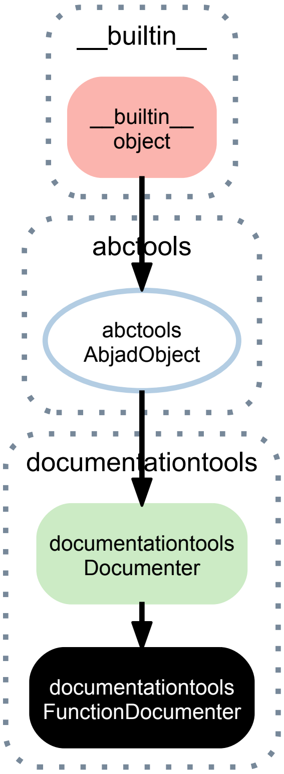
`FunctionCrawler.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`FunctionCrawler.__repr__()`
Interpreter representation of Abjad object.

Return string.

57.2.7 documentationtools.FunctionDocumenter



class `documentationtools.`**`FunctionDocumenter`**(*obj*, *prefix='abjad.tools.'*)

FunctionDocumenter generates an ReST entry for a given function:

```
>>> documenter = documentationtools.FunctionDocumenter(notetools.make_notes)
>>> print documenter()
notetools.make_notes
====================

.. autofunction:: abjad.tools.notetools.make_notes.make_notes
   :noindex:
```

Returns `FunctionDocumenter`` instance.

Read-only properties

`FunctionDocumenter.`**`module_name`**

`FunctionDocumenter.`**`object`**

`FunctionDocumenter.`**`prefix`**

`FunctionDocumenter.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`FunctionDocumenter.`**`__call__`**`()`

Generate documentation.

Returns string.

`FunctionDocumenter.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`FunctionDocumenter.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`FunctionDocumenter.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`FunctionDocumenter.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`FunctionDocumenter.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

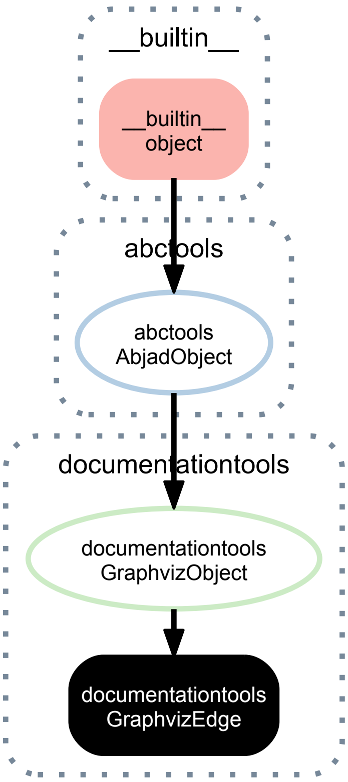
`FunctionDocumenter.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`FunctionDocumenter.__repr__()`
Interpreter representation of Abjad object.

Return string.

57.2.8 documentationtools.GraphvizEdge



class `documentationtools.`**`GraphvizEdge`**(*attributes=None*, *is_directed=True*)
A Graphviz edge.

Read-only properties

`GraphvizEdge.`**`attributes`**

`GraphvizEdge.`**`head`**

`GraphvizEdge.`**`storage_format`**
Storage format of Abjad object.

Return string.

`GraphvizEdge.`**`tail`**

Read/write properties

`GraphvizEdge.`**`is_directed`**

Special methods

`GraphvizEdge.`**`__call__`**(**args*)

`GraphvizEdge.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`GraphvizEdge.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`GraphvizEdge.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`GraphvizEdge.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`GraphvizEdge.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

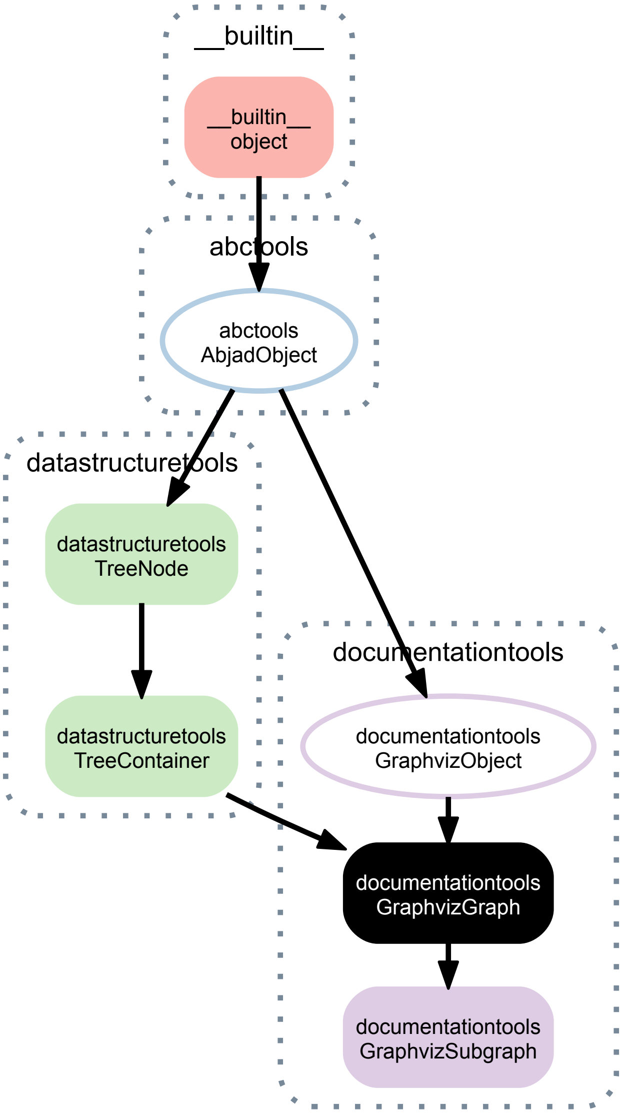
`GraphvizEdge.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`GraphvizEdge.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

57.2.9 documentationtools.GraphvizGraph



class documentationtools.**GraphvizGraph**(*attributes=None*, *children=None*, *edge_attributes=None*, *is_digraph=True*, *name=None*, *node_attributes=None*)

Abjad model of a Graphviz graph:

```
>>> graph = documentationtools.GraphvizGraph(name='G')
```

Create other graphviz objects to insert into the graph:

```
>>> cluster_0 = documentationtools.GraphvizSubgraph(name='0')
>>> cluster_1 = documentationtools.GraphvizSubgraph(name='1')
>>> a0 = documentationtools.GraphvizNode(name='a0')
>>> a1 = documentationtools.GraphvizNode(name='a1')
>>> a2 = documentationtools.GraphvizNode(name='a2')
>>> a3 = documentationtools.GraphvizNode(name='a3')
>>> b0 = documentationtools.GraphvizNode(name='b0')
>>> b1 = documentationtools.GraphvizNode(name='b1')
>>> b2 = documentationtools.GraphvizNode(name='b2')
>>> b3 = documentationtools.GraphvizNode(name='b3')
>>> start = documentationtools.GraphvizNode(name='start')
>>> end = documentationtools.GraphvizNode(name='end')
```

Group objects together into a tree:

```
>>> graph.extend([cluster_0, cluster_1, start, end])
>>> cluster_0.extend([a0, a1, a2, a3])
>>> cluster_1.extend([b0, b1, b2, b3])
```

Connect objects together with edges:

```
>>> edge = documentationtools.GraphvizEdge()(start, a0)
>>> edge = documentationtools.GraphvizEdge()(start, b0)
>>> edge = documentationtools.GraphvizEdge()(a0, a1)
>>> edge = documentationtools.GraphvizEdge()(a1, a2)
>>> edge = documentationtools.GraphvizEdge()(a1, b3)
>>> edge = documentationtools.GraphvizEdge()(a2, a3)
>>> edge = documentationtools.GraphvizEdge()(a3, a0)
>>> edge = documentationtools.GraphvizEdge()(a3, end)
>>> edge = documentationtools.GraphvizEdge()(b0, b1)
>>> edge = documentationtools.GraphvizEdge()(b1, b2)
>>> edge = documentationtools.GraphvizEdge()(b2, b3)
>>> edge = documentationtools.GraphvizEdge()(b2, a3)
>>> edge = documentationtools.GraphvizEdge()(b3, end)
```

Add attributes to style the objects:

```
>>> cluster_0.attributes['style'] = 'filled'
>>> cluster_0.attributes['color'] = 'lightgrey'
>>> cluster_0.attributes['label'] = 'process #1'
>>> cluster_0.node_attributes['style'] = 'filled'
>>> cluster_0.node_attributes['color'] = 'white'
>>> cluster_1.attributes['color'] = 'blue'
>>> cluster_1.attributes['label'] = 'process #2'
>>> cluster_1.node_attributes['style'] = ('filled', 'rounded')
>>> start.attributes['shape'] = 'Mdiamond'
>>> end.attributes['shape'] = 'Msquare'
```

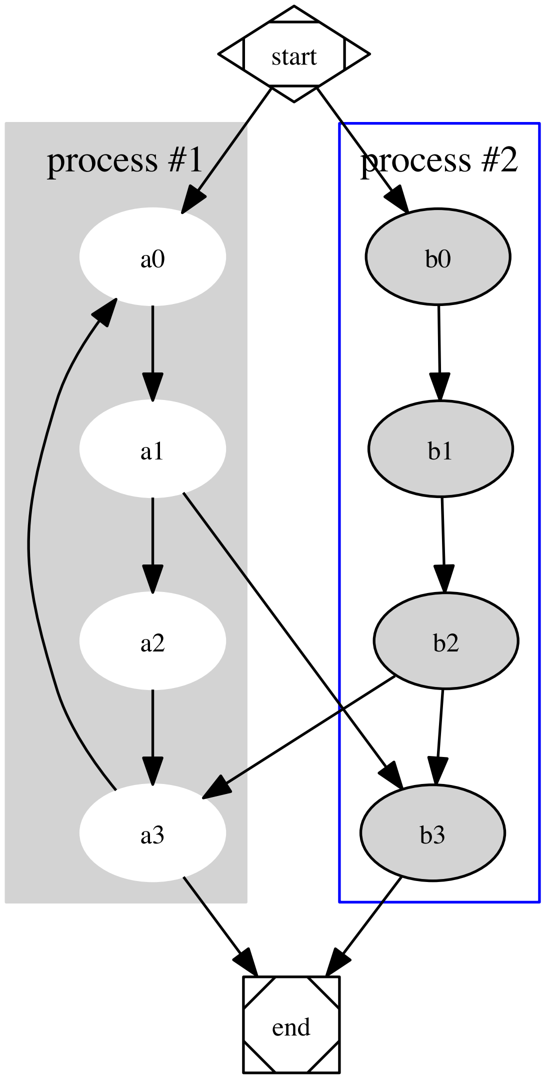
Access the computed graphviz format of the graph:

```
>>> print graph.graphviz_format
digraph G {
    subgraph cluster_0 {
        graph [color=lightgrey,
            label="process #1",
            style=filled];
        node [color=white,
            style=filled];
        a0;
        a1;
        a2;
        a3;
        a0 -> a1;
        a1 -> a2;
        a2 -> a3;
        a3 -> a0;
    }
    subgraph cluster_1 {
        graph [color=blue,
            label="process #2"];
        node [style="filled, rounded"];
        b0;
        b1;
        b2;
        b3;
        b0 -> b1;
        b1 -> b2;
        b2 -> b3;
    }
    start [shape=Mdiamond];
    end [shape=Msquare];
    a1 -> b3;
    a3 -> end;
    b2 -> a3;
    b3 -> end;
    start -> a0;
    start -> b0;
```

```
}
```

View the graph:

```
>>> iotools.graph(graph)
```



Graphs can also be created without defining names. Canonical names will be automatically determined for all members whose *name* is None:

```
>>> graph = documentationtools.GraphvizGraph()
>>> graph.append(documentationtools.GraphvizSubgraph())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizNode())
>>> graph[0].append(documentationtools.GraphvizSubgraph())
>>> graph[0][-1].append(documentationtools.GraphvizNode())
>>> graph.append(documentationtools.GraphvizNode())
>>> edge = documentationtools.GraphvizEdge()(graph[0][1], graph[1])
>>> edge = documentationtools.GraphvizEdge()(graph[0][0], graph[0][-1][0])
```

```
>>> print graph.graphviz_format
digraph Graph {
    subgraph cluster_0 {
        node_0_0;
        node_0_1;
        node_0_2;
        subgraph cluster_0_3 {
            node_0_3_0;
        }
```

```
            node_0_0 -> node_0_3_0;
        }
        node_1;
        node_0_1 -> node_1;
    }
```

Return GraphvizGraph instance.

Read-only properties

`GraphvizGraph.`**`attributes`**

`GraphvizGraph.`**`canonical_name`**

`GraphvizGraph.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`GraphvizGraph.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`GraphvizGraph.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

GraphvizGraph.**edge_attributes**

GraphvizGraph.**graph_order**

GraphvizGraph.**graphviz_format**

GraphvizGraph.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

GraphvizGraph.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

GraphvizGraph.**node_attributes**

GraphvizGraph.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

GraphvizGraph.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

GraphvizGraph.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

GraphvizGraph.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

GraphvizGraph.**storage_format**

Storage format of Abjad object.

Return string.

GraphvizGraph.**unflattened_graphviz_format**

Read/write properties

GraphvizGraph.**is_digraph**

GraphvizGraph.**name**

Methods

GraphvizGraph.**append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
```

```
            TreeNode(),
            TreeNode()
            )
    )
```

Return *None*.

`GraphvizGraph.`**`extend`**`(`*expr*`)`

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`GraphvizGraph.`**`index`**`(`*node*`)`

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`GraphvizGraph.`**`insert`**`(`*i*, *node*`)`

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
```

```
            TreeNode()
            )
        )
```

Return *None*.

GraphvizGraph.**pop**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

GraphvizGraph.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

GraphvizGraph.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

GraphvizGraph.**__copy__**(*args*)

GraphvizGraph.**__deepcopy__**(*args*)

GraphvizGraph.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

GraphvizGraph.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

GraphvizGraph.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizGraph.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

`GraphvizGraph.__getstate__()`

`GraphvizGraph.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`GraphvizGraph.__iter__()`

`GraphvizGraph.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`GraphvizGraph.__len__()`
Return nonnegative integer number of nodes in container.

`GraphvizGraph.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`GraphvizGraph.__ne__(expr)`

`GraphvizGraph.__repr__()`

`GraphvizGraph.__setitem__(i, expr)`
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

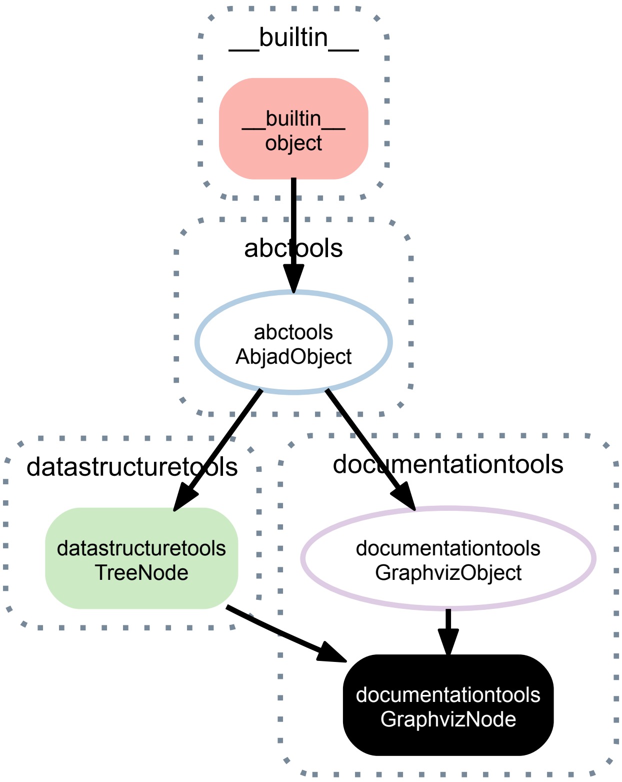
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

GraphvizGraph.**__setstate__**(*state*)

57.2.10 documentationtools.GraphvizNode



class documentationtools.**GraphvizNode**(*attributes=None*, *name=None*)

Read-only properties

GraphvizNode.**attributes**

GraphvizNode.**canonical_name**

GraphvizNode.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

GraphvizNode.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

GraphvizNode.**edges**

GraphvizNode.**graph_order**

GraphvizNode.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

GraphvizNode.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

GraphvizNode.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

GraphvizNode.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

GraphvizNode.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

GraphvizNode.**name**

Special methods

GraphvizNode.**__copy__**(**args*)

GraphvizNode.**__deepcopy__**(**args*)

GraphvizNode.**__eq__**(*expr*)

GraphvizNode.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizNode.**__getstate__**()

GraphvizNode.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GraphvizNode.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizNode.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

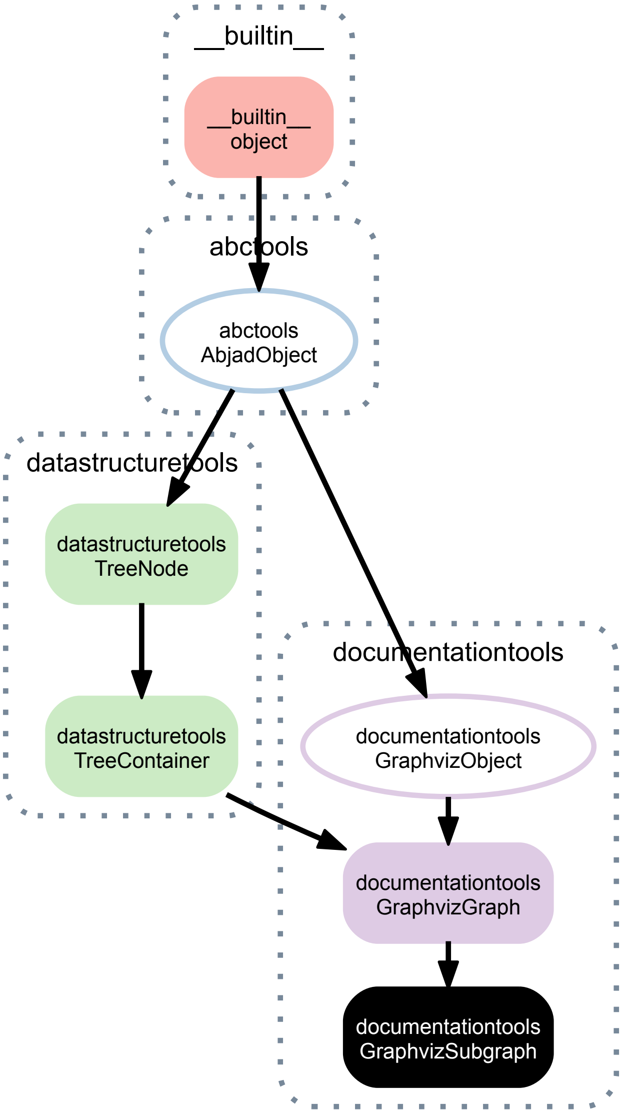
Raise exception.

GraphvizNode.**__ne__**(*expr*)

GraphvizNode.**__repr__**()

GraphvizNode.**__setstate__**(*state*)

57.2.11 documentationtools.GraphvizSubgraph



class documentationtools.**GraphvizSubgraph**(*attributes=None*, *children=None*, *edge_attributes=None*, *is_cluster=True*, *name=None*, *node_attributes=None*)

A Graphviz cluster subgraph.

Read-only properties

GraphvizSubgraph.**attributes**

GraphvizSubgraph.**canonical_name**

GraphvizSubgraph.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

GraphvizSubgraph.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

GraphvizSubgraph.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

GraphvizSubgraph.**edge_attributes**

GraphvizSubgraph.**edges**

GraphvizSubgraph.**graph_order**

GraphvizSubgraph.**graphviz_format**

GraphvizSubgraph.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

GraphvizSubgraph.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

GraphvizSubgraph.**node_attributes**

GraphvizSubgraph.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`GraphvizSubgraph.`**`parent`**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`GraphvizSubgraph.`**`proper_parentage`**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

`GraphvizSubgraph.`**`root`**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
```

```
>>> c.root is a
True
```

Return *TreeNode* instance.

GraphvizSubgraph.**storage_format**
Storage format of Abjad object.

Return string.

GraphvizSubgraph.**unflattened_graphviz_format**

Read/write properties

GraphvizSubgraph.**is_cluster**

GraphvizSubgraph.**is_digraph**

GraphvizSubgraph.**name**

Methods

GraphvizSubgraph.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

GraphvizSubgraph.**extend**(*expr*)
Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
```

```
        )
    )
```

Return *None*.

GraphvizSubgraph.**index**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

GraphvizSubgraph.**insert**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

GraphvizSubgraph.**pop**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

GraphvizSubgraph.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

GraphvizSubgraph.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

GraphvizSubgraph.**__copy__**(**args*)

GraphvizSubgraph.**__deepcopy__**(**args*)

GraphvizSubgraph.**__delitem__**(*i*)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

`GraphvizSubgraph.__eq__(expr)`
True if type, duration and children are equivalent, otherwise False.

Return boolean.

`GraphvizSubgraph.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`GraphvizSubgraph.__getitem__(i)`
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

GraphvizSubgraph.**__getstate__**()

GraphvizSubgraph.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GraphvizSubgraph.**__iter__**()

GraphvizSubgraph.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizSubgraph.**__len__**()
Return nonnegative integer number of nodes in container.

GraphvizSubgraph.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GraphvizSubgraph.**__ne__**(*expr*)

GraphvizSubgraph.**__repr__**()

GraphvizSubgraph.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

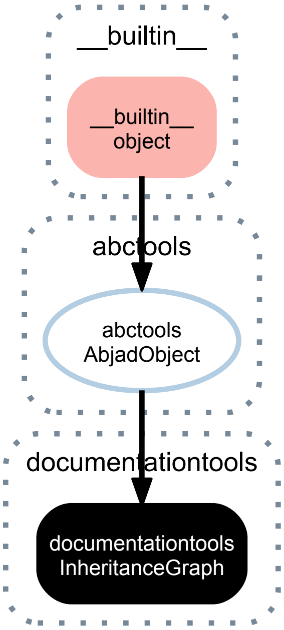
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

GraphvizSubgraph.**__setstate__**(*state*)

57.2.12 documentationtools.InheritanceGraph



class `documentationtools.`**`InheritanceGraph`**(*addresses=('abjad',), lineage_addresses=None, lineage_prune_distance=None, recurse_into_submodules=True, root_addresses=None, use_clusters=True, use_groups=True*)

Generates a graph of a class or collection of klasses as a dictionary of parent-children relationships:

```
>>> class A(object): pass
...
>>> class B(A): pass
...
>>> class C(B): pass
...
>>> class D(B): pass
...
>>> class E(C, D): pass
...
>>> class F(A): pass
...
```

```
>>> graph = documentationtools.InheritanceGraph(addresses=(F, E))
```

`InheritanceGraph` may be instantiated from one or more instances, klasses or modules. If instantiated from a module, all public klasses in that module will be taken into the graph.

A *root_class* keyword may be defined at instantiation, which filters out all klasses from the graph which do not inherit from that *root_class* (or are not already the *root_class*):

```
>>> graph = documentationtools.InheritanceGraph(
...     (A, B, C, D, E, F), root_addresses=(B,))
```

The class is intended for use in documenting packages.

To document all of Abjad, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',))
```

To document only those classes descending from Container, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     root_addresses=(Container,)
...     )
```

To document only those classes whose lineage pass through componenttools, use this formulation:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     lineage_addresses=(componenttools,),
...     )
```

When creating the Graphviz representation, classes in the inheritance graph may be hidden, based on their distance from any defined lineage class:

```
>>> graph = documentationtools.InheritanceGraph(
...     addresses=('abjad',),
...     lineage_addresses=(marktools.Mark,),
...     lineage_prune_distance=1,
...     )
```

Returns `InheritanceGraph` instance.

Read-only properties

`InheritanceGraph.`**`addresses`**

`InheritanceGraph.`**`child_parents_mapping`**

`InheritanceGraph.`**`graphviz_format`**

`InheritanceGraph.`**`graphviz_graph`**

`InheritanceGraph.`**`immediate_klasses`**

`InheritanceGraph.`**`lineage_addresses`**

`InheritanceGraph.`**`lineage_distance_mapping`**

`InheritanceGraph.`**`lineage_klasses`**

`InheritanceGraph.`**`lineage_prune_distance`**

`InheritanceGraph.`**`parent_children_mapping`**

`InheritanceGraph.`**`recurse_into_submodules`**

`InheritanceGraph.`**`root_addresses`**

`InheritanceGraph.`**`root_klasses`**

`InheritanceGraph.`**`storage_format`**
Storage format of Abjad object.

Return string.

`InheritanceGraph.`**`use_clusters`**

`InheritanceGraph.`**`use_groups`**

Special methods

`InheritanceGraph.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

InheritanceGraph.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InheritanceGraph.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

InheritanceGraph.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

InheritanceGraph.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

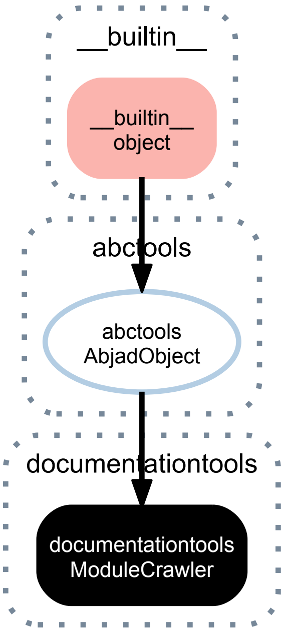
InheritanceGraph.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

InheritanceGraph.**__repr__**()
Interpreter representation of Abjad object.

Return string.

57.2.13 documentationtools.ModuleCrawler



class documentationtools.**ModuleCrawler**(*code_root='.', ignored_directories=['test', '.svn', '__pycache__'], root_package_name=None, visit_private_modules=False*)
Crawls *code_root*, yielding all module objects whose name begins with *root_package_name*.

Return *ModuleCrawler* instance.

Read-only properties

ModuleCrawler.**code_root**

ModuleCrawler.**ignored_directories**

ModuleCrawler.**root_package_name**

ModuleCrawler.**storage_format**
Storage format of Abjad object.

Return string.

ModuleCrawler.**visit_private_modules**

Special methods

ModuleCrawler.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ModuleCrawler.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ModuleCrawler.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ModuleCrawler.**__iter__**()

ModuleCrawler.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ModuleCrawler.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

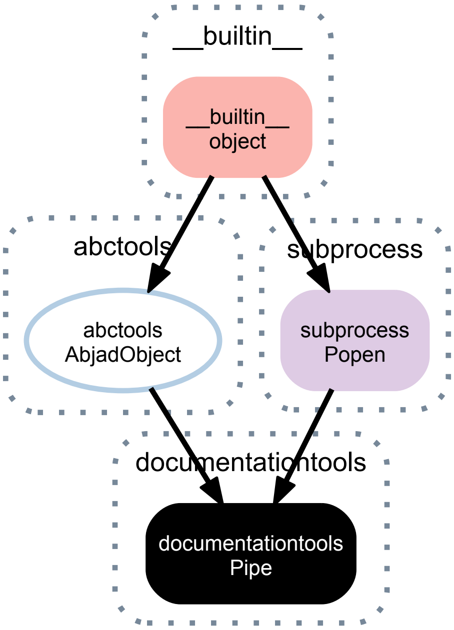
ModuleCrawler.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

ModuleCrawler.**__repr__**()
Interpreter representation of Abjad object.

Return string.

57.2.14 documentationtools.Pipe



class documentationtools.**Pipe**(*executable='python', arguments=['-i'], timeout=0*)
A two-way, non-blocking pipe for interprocess communication:

```
>>> pipe = documentationtools.Pipe('python', ['-i'])
>>> pipe.writeline('my_list = [1, 2, 3]')
>>> pipe.writeline('print my_list')
```

Return *Pipe* instance.

Read-only properties

Pipe.**arguments**

Pipe.**executable**

Pipe.**storage_format**
Storage format of Abjad object.

Return string.

Pipe.**timeout**

Methods

Pipe.**close**()
Close the pipe.

Pipe.**communicate**(*input=None*)
Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.

communicate() returns a tuple (stdout, stderr).

Pipe.**kill**()
Kill the process with SIGKILL

Pipe.**poll**()

`Pipe.`**`read`**`()`
Read from the pipe.

`Pipe.`**`read_wait`**`(`*seconds=0.01*`)`
Try to read from the pipe. Wait *seconds* if nothing comes out, and repeat.

Should be used with caution, as this may loop forever.

`Pipe.`**`send_signal`**`(`*sig*`)`
Send a signal to the process

`Pipe.`**`terminate`**`()`
Terminate the process with SIGTERM

`Pipe.`**`wait`**`()`
Wait for child process to terminate. Returns returncode attribute.

`Pipe.`**`write`**`(`*data*`)`
Write *data* into the pipe.

`Pipe.`**`write_line`**`(`*data*`)`
Write *data* into the pipe, terminated by a newline.

Special methods

`Pipe.`**`__del__`**`(`*_maxint=9223372036854775807*, *_active=*[ ]`)`

`Pipe.`**`__eq__`**`(`*expr*`)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`Pipe.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Pipe.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`Pipe.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`Pipe.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

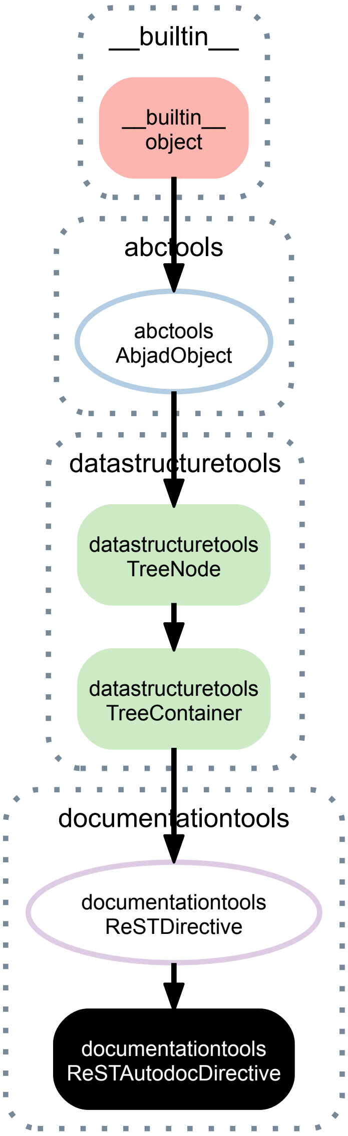
`Pipe.`**`__ne__`**`(`*expr*`)`
Defined equal to the opposite of equality.

Return boolean.

`Pipe.`**`__repr__`**`()`
Interpreter representation of Abjad object.

Return string.

57.2.15 documentationtools.ReSTAutodocDirective



class documentationtools.**ReSTAutodocDirective**(*argument=None*, *children=None*, *directive=None*, *name=None*, *options=None*)

An ReST autodoc directive:

```
>>> autodoc = documentationtools.ReSTAutodocDirective(
...     argument='abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner',
...     directive='autoclass',
...     )
>>> autodoc.options['noindex'] = True
>>> autodoc
ReSTAutodocDirective(
    argument='abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner',
    directive='autoclass',
    options={
        'noindex': True
        }
    )
```

```
>>> print autodoc.rest_format
.. autoclass:: abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner
   :noindex:
```

Return *ReSTAutodocDirective* instance.

Read-only properties

ReSTAutodocDirective.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

ReSTAutodocDirective.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTAutodocDirective.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
```

```
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTAutodocDirective.**graph_order**

ReSTAutodocDirective.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTAutodocDirective.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

ReSTAutodocDirective.**node_klass**

ReSTAutodocDirective.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

ReSTAutodocDirective.**options**

ReSTAutodocDirective.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTAutodocDirective.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTAutodocDirective.**rest_format**

ReSTAutodocDirective.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTAutodocDirective.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

ReSTAutodocDirective.**argument**

ReSTAutodocDirective.**directive**

ReSTAutodocDirective.**name**

Methods

ReSTAutodocDirective.**append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTAutodocDirective.**extend**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTAutodocDirective.**index**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

ReSTAutodocDirective.**insert**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTAutodocDirective.**pop**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTAutodocDirective.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTAutodocDirective.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTAutodocDirective.**__copy__**(**args*)

ReSTAutodocDirective.**__deepcopy__**(**args*)

ReSTAutodocDirective.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTAutodocDirective.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTAutodocDirective.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTAutodocDirective.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

ReSTAutodocDirective.**__getstate__**()

ReSTAutodocDirective.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTAutodocDirective.**__iter__**()

ReSTAutodocDirective.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTAutodocDirective.**__len__**()
Return nonnegative integer number of nodes in container.

ReSTAutodocDirective.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTAutodocDirective.**__ne__**(*expr*)

ReSTAutodocDirective.**__repr__**()

ReSTAutodocDirective.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

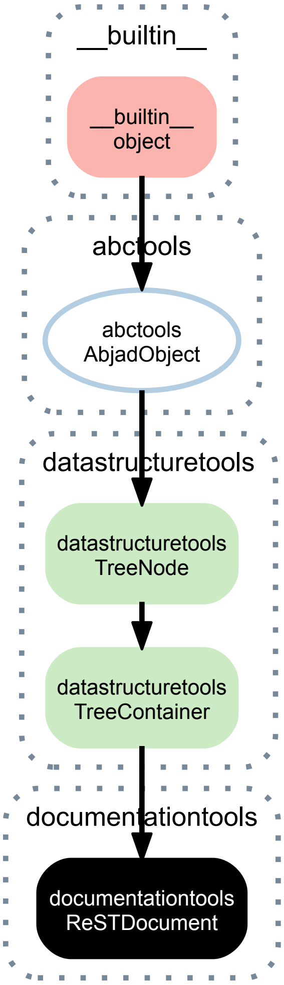
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

ReSTAutodocDirective.**__setstate__**(*state*)

57.2.16 documentationtools.ReSTDocument



class documentationtools.**ReSTDocument**(*children=None*, *name=None*)

An ReST document tree:

```
>>> document = documentationtools.ReSTDocument()
>>> document
ReSTDocument()
```

```
>>> document.append(documentationtools.ReSTHeading(level=0, text='Hello World!'))
>>> document.append(documentationtools.ReSTParagraph(text='blah blah blah'))
>>> toc = documentationtools.ReSTTOCDirective()
>>> toc.append('foo/bar')
>>> toc.append('bar/baz')
>>> toc.append('quux')
>>> document.append(toc)
```

```
>>> document
ReSTDocument(
    children=(
        ReSTHeading(
            level=0,
            text='Hello World!'
```

```
            ),
        ReSTParagraph(
            text='blah blah blah',
            wrap=True
            ),
        ReSTTOCDirective(
            children=(
                ReSTTOCItem(
                    text='foo/bar'
                    ),
                ReSTTOCItem(
                    text='bar/baz'
                    ),
                ReSTTOCItem(
                    text='quux'
                    )
                )
            )
        )
    )
```

```
>>> print document.rest_format
############
Hello World!
############

blah blah blah

.. toctree::

   foo/bar
   bar/baz
   quux
```

Return *ReSTDocument* instance.

Read-only properties

ReSTDocument.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

ReSTDocument.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTDocument.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTDocument.**graph_order**

ReSTDocument.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTDocument.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

ReSTDocument.**node_klass**

ReSTDocument.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

ReSTDocument.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTDocument.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTDocument.**rest_format**

ReSTDocument.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTDocument.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

ReSTDocument.**name**

Methods

ReSTDocument.**append**(*node*)
Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTDocument.`**`extend`**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTDocument.`**`index`**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`ReSTDocument.`**`insert`**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
```

```
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTDocument.pop(i=-1)`
Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

`ReSTDocument.remove(node)`
Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
```

```
            TreeNode()
            )
        )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTDocument.**__contains__**(*expr*)
True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTDocument.**__copy__**(**args*)

ReSTDocument.**__deepcopy__**(**args*)

ReSTDocument.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTDocument.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTDocument.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTDocument.**__getitem__**(*i*)

Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

ReSTDocument.**__getstate__**()

ReSTDocument.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

ReSTDocument.**__iter__**()

ReSTDocument.**__le__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ReSTDocument.**__len__**()

Return nonnegative integer number of nodes in container.

ReSTDocument.**__lt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ReSTDocument.**__ne__**(*expr*)

ReSTDocument.**__repr__**()

ReSTDocument.**__setitem__**(*i*, *expr*)

Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

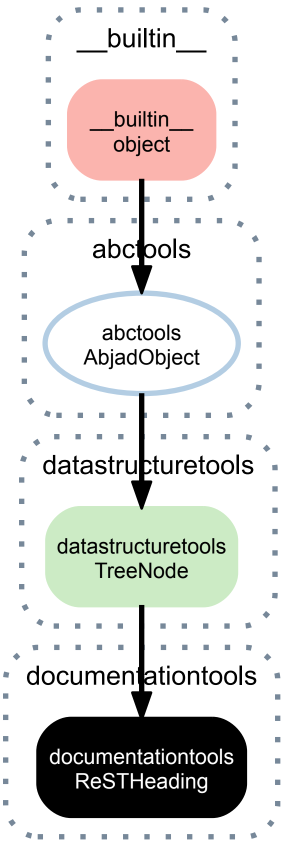
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

ReSTDocument.**__setstate__**(*state*)

57.2.17 documentationtools.ReSTHeading



class documentationtools.**ReSTHeading**(*level=0*, *name=None*, *text=''*)

An ReST Heading:

```
>>> heading = documentationtools.ReSTHeading(level=2, text='Section A')
>>> heading
ReSTHeading(
    level=2,
    text='Section A'
    )
```

```
>>> print heading.rest_format
Section A
=========
```

Return *ReSTHeading* instance.

Read-only properties

ReSTHeading.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTHeading.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTHeading.**graph_order**

ReSTHeading.**heading_characters**

ReSTHeading.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTHeading.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTHeading.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTHeading.**rest_format**

ReSTHeading.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTHeading.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

ReSTHeading.**level**

ReSTHeading.**name**

ReSTHeading.**text**

Special methods

ReSTHeading.**__copy__**(**args*)

ReSTHeading.**__deepcopy__**(**args*)

ReSTHeading.**__eq__**(*expr*)

ReSTHeading.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTHeading.**__getstate__**()

ReSTHeading.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTHeading.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTHeading.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

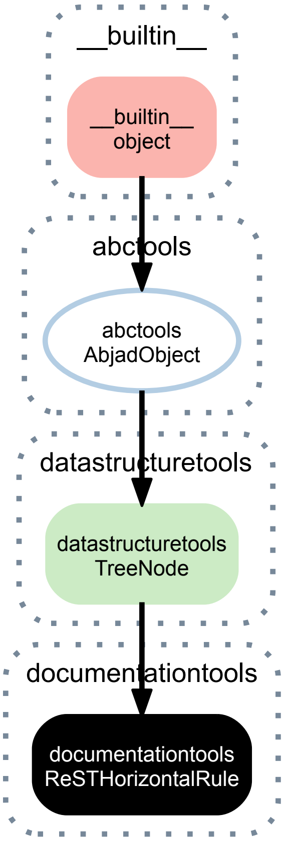
Raise exception.

ReSTHeading.**__ne__**(*expr*)

ReSTHeading.**__repr__**()

ReSTHeading.**__setstate__**(*state*)

57.2.18 documentationtools.ReSTHorizontalRule



class documentationtools.**ReSTHorizontalRule**(*name=None*)
An ReST horizontal rule:

```
>>> rule = documentationtools.ReSTHorizontalRule()
>>> rule
ReSTHorizontalRule()
```

```
>>> print rule.rest_format
--------
```

Return *ReSTHorizontalRule* instance.

Read-only properties

ReSTHorizontalRule.**depth**
The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTHorizontalRule.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTHorizontalRule.**graph_order**

ReSTHorizontalRule.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTHorizontalRule.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`ReSTHorizontalRule.`**`proper_parentage`**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

`ReSTHorizontalRule.`**`rest_format`**

`ReSTHorizontalRule.`**`root`**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

`ReSTHorizontalRule.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`ReSTHorizontalRule.`**`name`**

Special methods

`ReSTHorizontalRule.`**`__copy__`**`(*args)`

ReSTHorizontalRule.**__deepcopy__**(**args*)

ReSTHorizontalRule.**__eq__**(*expr*)

ReSTHorizontalRule.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTHorizontalRule.**__getstate__**()

ReSTHorizontalRule.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTHorizontalRule.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTHorizontalRule.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

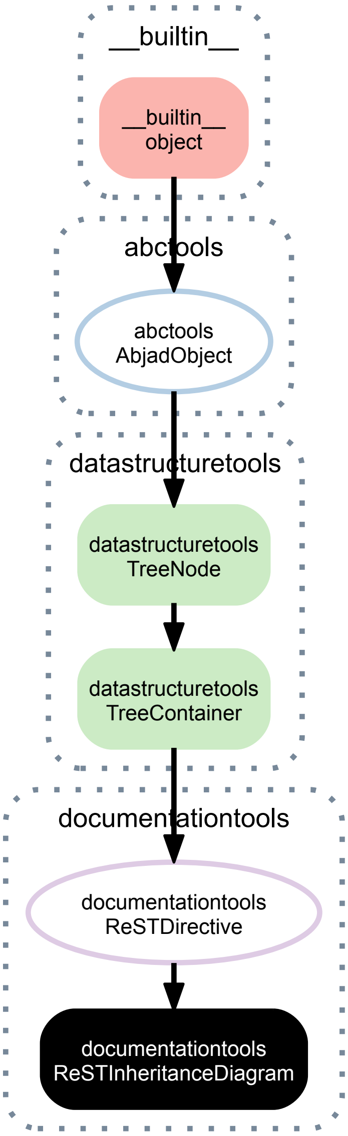
Raise exception.

ReSTHorizontalRule.**__ne__**(*expr*)

ReSTHorizontalRule.**__repr__**()

ReSTHorizontalRule.**__setstate__**(*state*)

57.2.19 documentationtools.ReSTInheritanceDiagram



class `documentationtools.`**`ReSTInheritanceDiagram`**(*argument=None*, *children=None*, *name=None*, *options=None*)

An ReST inheritance diagram directive:

```
>>> documentationtools.ReSTInheritanceDiagram(argument=beamtools.BeamSpanner)
ReSTInheritanceDiagram(
    argument='abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner',
    options={
        'private-bases': True
        }
    )
```

```
>>> print _.rest_format
.. inheritance-diagram:: abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner
   :private-bases:
```

Return *ReSTInheritanceDiagram* instance.

Read-only properties

`ReSTInheritanceDiagram.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`ReSTInheritanceDiagram.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`ReSTInheritanceDiagram.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
```

```
f
g
```

Return dictionary.

ReSTInheritanceDiagram.**directive**

ReSTInheritanceDiagram.**graph_order**

ReSTInheritanceDiagram.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTInheritanceDiagram.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

ReSTInheritanceDiagram.**node_klass**

ReSTInheritanceDiagram.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

ReSTInheritanceDiagram.**options**

ReSTInheritanceDiagram.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTInheritanceDiagram.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTInheritanceDiagram.**rest_format**

ReSTInheritanceDiagram.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

`ReSTInheritanceDiagram.`**`storage_format`**

Storage format of Abjad object.

Return string.

Read/write properties

`ReSTInheritanceDiagram.`**`argument`**

`ReSTInheritanceDiagram.`**`name`**

Methods

`ReSTInheritanceDiagram.`**`append`**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTInheritanceDiagram.`**`extend`**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTInheritanceDiagram.`**`index`**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`ReSTInheritanceDiagram.`**`insert`**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTInheritanceDiagram.`**`pop`**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTInheritanceDiagram.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTInheritanceDiagram.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTInheritanceDiagram.**__copy__**(**args*)

ReSTInheritanceDiagram.**__deepcopy__**(**args*)

ReSTInheritanceDiagram.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTInheritanceDiagram.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTInheritanceDiagram.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTInheritanceDiagram.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

ReSTInheritanceDiagram.**__getstate__**()

ReSTInheritanceDiagram.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTInheritanceDiagram.**__iter__**()

ReSTInheritanceDiagram.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTInheritanceDiagram.**__len__**()
Return nonnegative integer number of nodes in container.

ReSTInheritanceDiagram.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTInheritanceDiagram.**__ne__**(*expr*)

ReSTInheritanceDiagram.**__repr__**()

ReSTInheritanceDiagram.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

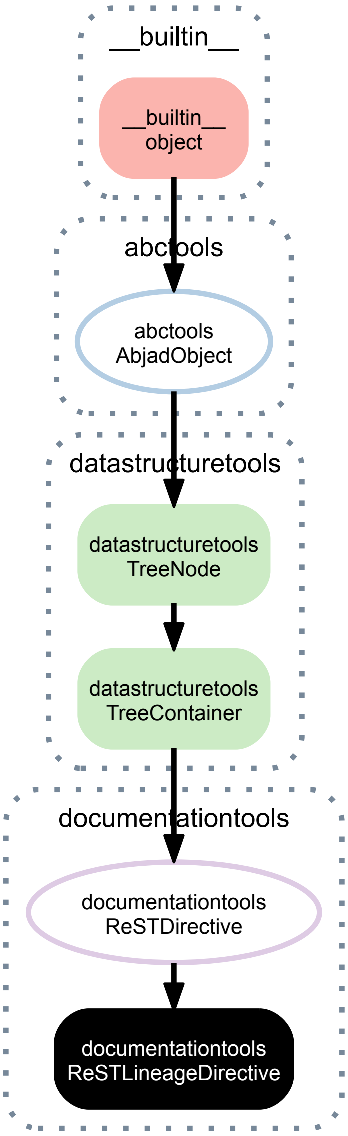
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

ReSTInheritanceDiagram.**__setstate__**(*state*)

57.2.20 documentationtools.ReSTLineageDirective



class `documentationtools.`**`ReSTLineageDirective`**(*argument=None*, *children=None*, *name=None*, *options=None*)

An ReST Abjad lineage diagram directive:

```
>>> documentationtools.ReSTLineageDirective(argument=beamtools.BeamSpanner)
ReSTLineageDirective(
    argument='abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner'
    )
```

```
>>> print _.rest_format
.. abjad-lineage:: abjad.tools.beamtools.BeamSpanner.BeamSpanner.BeamSpanner
```

Return *ReSTLineageDirective* instance.

Read-only properties

`ReSTLineageDirective.`**`children`**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

`ReSTLineageDirective.`**`depth`**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

`ReSTLineageDirective.`**`depthwise_inventory`**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

`ReSTLineageDirective.`**directive**

`ReSTLineageDirective.`**graph_order**

`ReSTLineageDirective.`**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

`ReSTLineageDirective.`**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

`ReSTLineageDirective.`**node_klass**

`ReSTLineageDirective.`**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

ReSTLineageDirective.**options**

ReSTLineageDirective.**parent**
The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTLineageDirective.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTLineageDirective.**rest_format**

ReSTLineageDirective.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTLineageDirective.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

ReSTLineageDirective.**argument**

ReSTLineageDirective.**name**

Methods

ReSTLineageDirective.**append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTLineageDirective.**extend**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTLineageDirective.`**`index`**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`ReSTLineageDirective.`**`insert`**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTLineageDirective.`**`pop`**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTLineageDirective.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTLineageDirective.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTLineageDirective.**__copy__**(**args*)

ReSTLineageDirective.**__deepcopy__**(**args*)

ReSTLineageDirective.**__delitem__**(*i*)

Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTLineageDirective.**__eq__**(*expr*)

True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTLineageDirective.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

ReSTLineageDirective.**__getitem__**(*i*)

Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

`ReSTLineageDirective.__getstate__()`

`ReSTLineageDirective.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`ReSTLineageDirective.__iter__()`

`ReSTLineageDirective.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ReSTLineageDirective.__len__()`
Return nonnegative integer number of nodes in container.

`ReSTLineageDirective.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ReSTLineageDirective.__ne__(expr)`

`ReSTLineageDirective.__repr__()`

`ReSTLineageDirective.__setitem__(i, expr)`
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

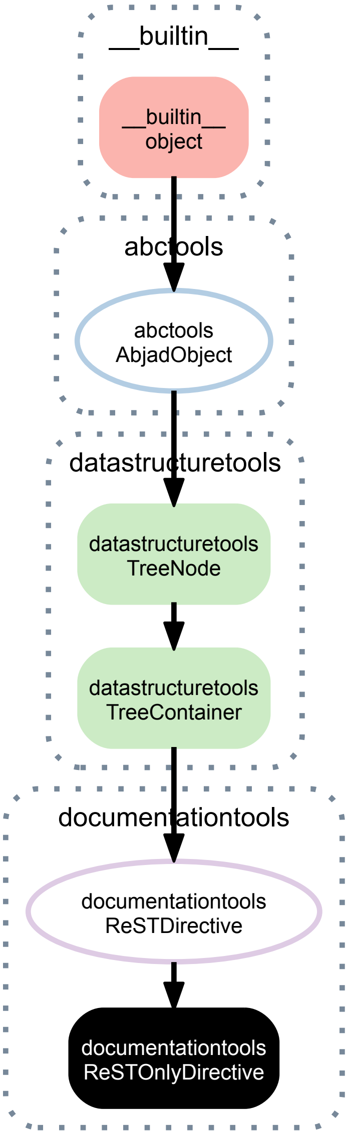
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

`ReSTLineageDirective.__setstate__(state)`

57.2.21 documentationtools.ReSTOnlyDirective



class documentationtools.**ReSTOnlyDirective**(*argument=None*, *children=None*, *name=None*)

An ReST *only* directive:

```
>>> only = documentationtools.ReSTOnlyDirective(argument='latex')
```

```
>>> heading = documentationtools.ReSTHeading(level=3, text='A LaTeX-Only Heading')
>>> only.append(heading)
>>> only
ReSTOnlyDirective(
    argument='latex',
    children=(
        ReSTHeading(
            level=3,
            text='A LaTeX-Only Heading'
            ),
        )
    )
```

```
>>> print only.rest_format
.. only:: latex

   A LaTeX-Only Heading
   --------------------
```

Return *ReSTOnlyDirective* instance.

Read-only properties

ReSTOnlyDirective.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

ReSTOnlyDirective.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTOnlyDirective.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
```

```
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTOnlyDirective.**directive**

ReSTOnlyDirective.**graph_order**

ReSTOnlyDirective.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTOnlyDirective.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
e
c
```

Return tuple.

ReSTOnlyDirective.**node_klass**

ReSTOnlyDirective.**nodes**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

ReSTOnlyDirective.**options**

ReSTOnlyDirective.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTOnlyDirective.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTOnlyDirective.**rest_format**

ReSTOnlyDirective.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTOnlyDirective.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

ReSTOnlyDirective.**argument**

ReSTOnlyDirective.**name**

Methods

ReSTOnlyDirective.**append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTOnlyDirective.**extend**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTOnlyDirective.`**`index`**(*node*)
Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`ReSTOnlyDirective.`**`insert`**(*i*, *node*)
Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTOnlyDirective.`**`pop`**(*i=-1*)
Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTOnlyDirective.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTOnlyDirective.**__contains__**(*expr*)

True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTOnlyDirective.**__copy__**(**args*)

ReSTOnlyDirective.**__deepcopy__**(**args*)

ReSTOnlyDirective.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTOnlyDirective.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTOnlyDirective.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOnlyDirective.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

ReSTOnlyDirective.**__getstate__**()

ReSTOnlyDirective.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTOnlyDirective.**__iter__**()

ReSTOnlyDirective.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOnlyDirective.**__len__**()
Return nonnegative integer number of nodes in container.

ReSTOnlyDirective.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTOnlyDirective.**__ne__**(*expr*)

ReSTOnlyDirective.**__repr__**()

ReSTOnlyDirective.**__setitem__**(*i*, *expr*)
Set *expr* in self at nonnegative integer index *i*, or set *expr* in self at slice i. Replace contents of *self[i]* with *expr*. Attach parentage to contents of *expr*, and detach parentage of any replaced nodes.

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.parent is a
True
```

```
>>> a.children == (b,)
True
```

```
>>> a[0] = c
```

```
>>> c.parent is a
True
```

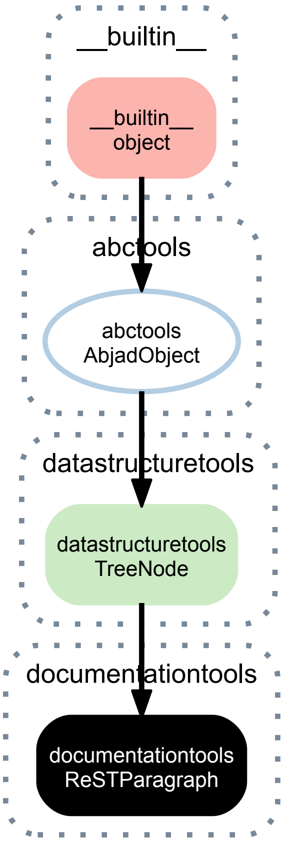
```
>>> b.parent is None
True
```

```
>>> a.children == (c,)
True
```

Return *None*.

ReSTOnlyDirective.**__setstate__**(*state*)

57.2.22 documentationtools.ReSTParagraph



class documentationtools.**ReSTParagraph**(*name=None*, *text=''*, *wrap=True*)

An ReST paragraph:

```
>>> paragraph = documentationtools.ReSTParagraph(text='blah blah blah')
>>> paragraph
ReSTParagraph(
    text='blah blah blah',
    wrap=True
    )
```

```
>>> print _.rest_format
blah blah blah
```

Handles automatic linewrapping.

Return *ReSTParagraph* instance.

Read-only properties

ReSTParagraph.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
```

```
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTParagraph.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTParagraph.**graph_order**

ReSTParagraph.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTParagraph.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTParagraph.**proper_parentage**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTParagraph.**rest_format**

ReSTParagraph.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

`ReSTParagraph.`**`storage_format`**
Storage format of Abjad object.

Return string.

Read/write properties

`ReSTParagraph.`**`name`**

`ReSTParagraph.`**`text`**

`ReSTParagraph.`**`wrap`**

Special methods

`ReSTParagraph.`**`__copy__`**`(`*`*args`*`)`

`ReSTParagraph.`**`__deepcopy__`**`(`*`*args`*`)`

`ReSTParagraph.`**`__eq__`**`(`*expr*`)`

`ReSTParagraph.`**`__ge__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`ReSTParagraph.`**`__getstate__`**`()`

`ReSTParagraph.`**`__gt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception

`ReSTParagraph.`**`__le__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

Raise exception.

`ReSTParagraph.`**`__lt__`**`(`*expr*`)`
Abjad objects by default do not implement this method.

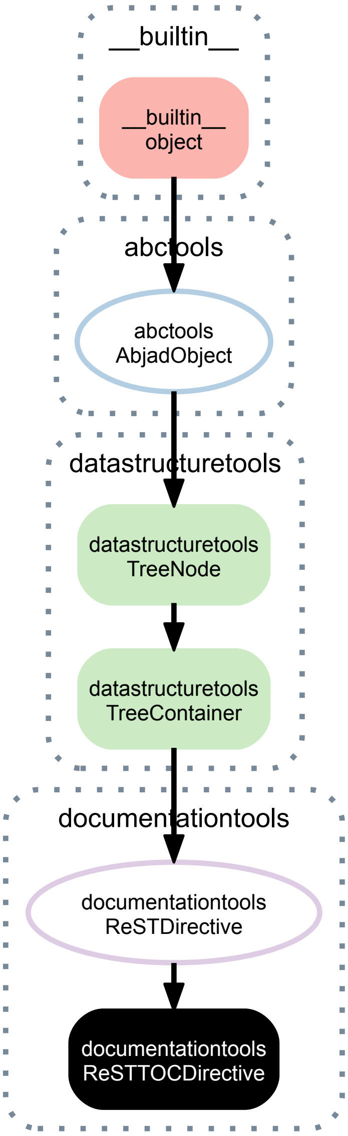
Raise exception.

`ReSTParagraph.`**`__ne__`**`(`*expr*`)`

`ReSTParagraph.`**`__repr__`**`()`

`ReSTParagraph.`**`__setstate__`**`(`*state*`)`

57.2.23 documentationtools.ReSTTOCDirective



class `documentationtools.`**`ReSTTOCDirective`**(*argument=None*, *children=None*, *name=None*, *options=None*)

An ReST TOCTree directive:

```
>>> toc = documentationtools.ReSTTOCDirective()
>>> for item in ['foo/index', 'bar/index', 'baz/index']:
...     toc.append(documentationtools.ReSTTOCItem(text=item))
...
>>> toc.options['maxdepth'] = 1
>>> toc.options['hidden'] = True
>>> toc
ReSTTOCDirective(
    children=(
        ReSTTOCItem(
            text='foo/index'
            ),
        ReSTTOCItem(
            text='bar/index'
            ),
        ReSTTOCItem(
            text='baz/index'
            )
        ),
    options={
```

```
            'hidden': True,
            'maxdepth': 1
            }
    )
```

```
>>> print toc.rest_format
.. toctree::
   :hidden:
   :maxdepth: 1

   foo/index
   bar/index
   baz/index
```

Return *ReSTTOCDirective* instance.

Read-only properties

ReSTTOCDirective.**children**

The children of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> a.children == (b, c)
True
```

```
>>> b.children == (d, e)
True
```

```
>>> e.children == ()
True
```

Return tuple of *TreeNode* instances.

ReSTTOCDirective.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTTOCDirective.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
```

```
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTTOCDirective.**directive**

ReSTTOCDirective.**graph_order**

ReSTTOCDirective.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTTOCDirective.**leaves**

The leaves of a *TreeContainer* instance:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeNode(name='c')
>>> d = datastructuretools.TreeNode(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> for leaf in a.leaves:
...     print leaf.name
...
d
```

```
e
c
```

Return tuple.

`ReSTTOCDirective.`**`node_klass`**

`ReSTTOCDirective.`**`nodes`**

The collection of *TreeNodes* produced by iterating a node depth-first:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeContainer()
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
```

```
>>> nodes = a.nodes
>>> len(nodes)
5
```

```
>>> nodes[0] is a
True
```

```
>>> nodes[1] is b
True
```

```
>>> nodes[2] is d
True
```

```
>>> nodes[3] is e
True
```

```
>>> nodes[4] is c
True
```

Return tuple.

`ReSTTOCDirective.`**`options`**

`ReSTTOCDirective.`**`parent`**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

`ReSTTOCDirective.`**`proper_parentage`**

The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTTOCDirective.**rest_format**

ReSTTOCDirective.**root**

The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTTOCDirective.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

ReSTTOCDirective.**argument**

ReSTTOCDirective.**name**

Methods

ReSTTOCDirective.**append**(*node*)

Append *node* to container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.append(b)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

```
>>> a.append(c)
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTTOCDirective.`**`extend`**(*expr*)

Extend *expr* against container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a
TreeContainer()
```

```
>>> a.extend([b, c])
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

`ReSTTOCDirective.`**`index`**(*node*)

Index *node* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a.index(b)
0
```

```
>>> a.index(c)
1
```

Return nonnegative integer.

`ReSTTOCDirective.`**`insert`**(*i*, *node*)

Insert *node* in container at index *i*:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
>>> d = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.insert(1, d)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode(),
        TreeNode()
        )
    )
```

Return *None*.

ReSTTOCDirective.**pop**(*i=-1*)

Pop node at index *i* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> node = a.pop()
```

```
>>> node == c
True
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return node.

ReSTTOCDirective.**remove**(*node*)

Remove *node* from container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c])
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        TreeNode()
        )
    )
```

```
>>> a.remove(b)
```

```
>>> a
TreeContainer(
    children=(
        TreeNode(),
        )
    )
```

Return *None*.

Special methods

ReSTTOCDirective.**__contains__**(*expr*)
True if expr is in container, otherwise False:

```
>>> container = datastructuretools.TreeContainer()
>>> a = datastructuretools.TreeNode()
>>> b = datastructuretools.TreeNode()
>>> container.append(a)
```

```
>>> a in container
True
```

```
>>> b in container
False
```

Return boolean.

ReSTTOCDirective.**__copy__**(**args*)

ReSTTOCDirective.**__deepcopy__**(**args*)

ReSTTOCDirective.**__delitem__**(*i*)
Find node at index or slice *i* in container and detach from parentage:

```
>>> container = datastructuretools.TreeContainer()
>>> leaf = datastructuretools.TreeNode()
```

```
>>> container.append(leaf)
>>> container.children == (leaf,)
True
```

```
>>> leaf.parent is container
True
```

```
>>> del(container[0])
```

```
>>> container.children == ()
True
```

```
>>> leaf.parent is None
True
```

Return *None*.

ReSTTOCDirective.**__eq__**(*expr*)
True if type, duration and children are equivalent, otherwise False.

Return boolean.

ReSTTOCDirective.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTTOCDirective.**__getitem__**(*i*)
Return node at index *i* in container:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeNode()
>>> c = datastructuretools.TreeContainer()
>>> d = datastructuretools.TreeNode()
>>> e = datastructuretools.TreeNode()
>>> f = datastructuretools.TreeNode()
```

```
>>> a.extend([b, c, f])
>>> c.extend([d, e])
```

```
>>> a[0] is b
True
```

```
>>> a[1] is c
True
```

```
>>> a[2] is f
True
```

If *i* is a string, the container will attempt to return the single child node, at any depth, whose *name* matches *i*:

```
>>> foo = datastructuretools.TreeContainer(name='foo')
>>> bar = datastructuretools.TreeContainer(name='bar')
>>> baz = datastructuretools.TreeNode(name='baz')
>>> quux = datastructuretools.TreeNode(name='quux')
```

```
>>> foo.append(bar)
>>> bar.extend([baz, quux])
```

```
>>> foo['bar'] is bar
True
```

```
>>> foo['baz'] is baz
True
```

```
>>> foo['quux'] is quux
True
```

Return *TreeNode* instance.

ReSTTOCDirective.**__getstate__**()

ReSTTOCDirective.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTTOCDirective.**__iter__**()

ReSTTOCDirective.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTTOCDirective.**__len__**()
Return nonnegative integer number of nodes in container.

ReSTTOCDirective.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

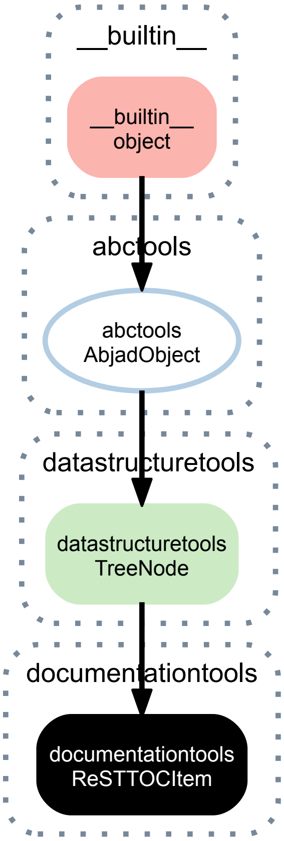
ReSTTOCDirective.**__ne__**(*expr*)

ReSTTOCDirective.**__repr__**()

ReSTTOCDirective.**__setitem__**(*i*, *expr*)

ReSTTOCDirective.**__setstate__**(*state*)

57.2.24 documentationtools.ReSTTOCItem



class documentationtools.**ReSTTOCItem**(*name=None*, *text=None*)

An ReST TOC item:

```
>>> item = documentationtools.ReSTTOCItem(text='api/index')
>>> item
ReSTTOCItem(
    text='api/index'
    )
```

```
>>> print item.rest_format
api/index
```

Return *ReSTTOCItem* instance.

Read-only properties

ReSTTOCItem.**depth**

The depth of a node in a rhythm-tree structure:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.depth
0
```

```
>>> a[0].depth
1
```

```
>>> a[0][0].depth
2
```

Return int.

ReSTTOCItem.**depthwise_inventory**

A dictionary of all nodes in a rhythm-tree, organized by their depth relative the root node:

```
>>> a = datastructuretools.TreeContainer(name='a')
>>> b = datastructuretools.TreeContainer(name='b')
>>> c = datastructuretools.TreeContainer(name='c')
>>> d = datastructuretools.TreeContainer(name='d')
>>> e = datastructuretools.TreeContainer(name='e')
>>> f = datastructuretools.TreeContainer(name='f')
>>> g = datastructuretools.TreeContainer(name='g')
```

```
>>> a.extend([b, c])
>>> b.extend([d, e])
>>> c.extend([f, g])
```

```
>>> inventory = a.depthwise_inventory
>>> for depth in sorted(inventory):
...     print 'DEPTH: {}'.format(depth)
...     for node in inventory[depth]:
...         print node.name
...
DEPTH: 0
a
DEPTH: 1
b
c
DEPTH: 2
d
e
f
g
```

Return dictionary.

ReSTTOCItem.**graph_order**

ReSTTOCItem.**improper_parentage**

The improper parentage of a node in a rhythm-tree, being the sequence of node beginning with itself and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.improper_parentage == (a,)
True
```

```
>>> b.improper_parentage == (b, a)
True
```

```
>>> c.improper_parentage == (c, b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTTOCItem.**parent**

The node's parent node:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.parent is None
True
```

```
>>> b.parent is a
True
```

```
>>> c.parent is b
True
```

Return *TreeNode* instance.

ReSTTOCItem.**proper_parentage**
The proper parentage of a node in a rhythm-tree, being the sequence of node beginning with the node's immediate parent and ending with the root node of the tree:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.proper_parentage == ()
True
```

```
>>> b.proper_parentage == (a,)
True
```

```
>>> c.proper_parentage == (b, a)
True
```

Return tuple of *TreeNode* instances.

ReSTTOCItem.**rest_format**

ReSTTOCItem.**root**
The root node of the tree: that node in the tree which has no parent:

```
>>> a = datastructuretools.TreeContainer()
>>> b = datastructuretools.TreeContainer()
>>> c = datastructuretools.TreeNode()
```

```
>>> a.append(b)
>>> b.append(c)
```

```
>>> a.root is a
True
>>> b.root is a
True
>>> c.root is a
True
```

Return *TreeNode* instance.

ReSTTOCItem.**storage_format**
Storage format of Abjad object.

Return string.

Read/write properties

ReSTTOCItem.**name**

ReSTTOCItem.**text**

Special methods

ReSTTOCItem.**__copy__**(**args*)

ReSTTOCItem.**__deepcopy__**(**args*)

ReSTTOCItem.**__eq__**(*expr*)

ReSTTOCItem.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTTOCItem.**__getstate__**()

ReSTTOCItem.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

ReSTTOCItem.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTTOCItem.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

ReSTTOCItem.**__ne__**(*expr*)

ReSTTOCItem.**__repr__**()

ReSTTOCItem.**__setstate__**(*state*)

57.3 Functions

57.3.1 documentationtools.compare_images

documentationtools.**compare_images**(*image_one*, *image_two*)
Compare *image_one* against *image_two* using ImageMagick's *compare* commandline tool.

Return *True* if images are the same, otherwise *False*.

If *compare* is not available, return *False*.

57.3.2 documentationtools.make_ligeti_example_lilypond_file

documentationtools.**make_ligeti_example_lilypond_file**(*music=None*)
New in version 2.9. Make Ligeti example LilyPond file.

Return LilyPond file.

57.3.3 documentationtools.make_reference_manual_graphviz_graph

documentationtools.**make_reference_manual_graphviz_graph**(*graph*)
Make a GraphvizGraph instance suitable for use in the Abjad reference manual.

Return GraphvizGraph instance.

57.3.4 documentationtools.make_reference_manual_lilypond_file

documentationtools.**make_reference_manual_lilypond_file**(*music=None*)

New in version 2.9. Make reference manual LilyPond file.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_reference_manual_lilypond_file(score)
```

```
>>> f(lilypond_file)

\version "2.15.37"
\language "english"
\include "/Users/trevorbaca/Documents/abjad/trunk/abjad/cfg/abjad.scm"

\layout {
    indent = #0
    ragged-right = ##t
    \context {
        \Score
        \remove Bar_number_engraver
        \override SpacingSpanner #'strict-grace-spacing = ##t
        \override SpacingSpanner #'strict-note-spacing = ##t
        \override SpacingSpanner #'uniform-stretching = ##t
        \override TupletBracket #'bracket-visibility = ##t
        \override TupletBracket #'minimum-length = #3
        \override TupletBracket #'padding = #2
        \override TupletBracket #'springs-and-rods = #ly:spanner::set-spacing-rods
        \override TupletNumber #'text = #tuplet-number::calc-fraction-text
        proportionalNotationDuration = #(ly:make-moment 1 32)
        tupletFullLength = True
    }
}

\paper {
    left-margin = 1.0\in
}

\score {
    \new Score <<
        \new Staff {
            c4
            d4
            e4
            f4
        }
    >>
}
```

Return LilyPond file.

57.3.5 documentationtools.make_text_alignment_example_lilypond_file

documentationtools.**make_text_alignment_example_lilypond_file**(*music=None*)

New in version 2.9. Make text-alignment example LilyPond file with *music*.

```
>>> score = Score([Staff('c d e f')])
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(score)
```

```
>>> f(lilypond_file)
% Abjad revision 5651
% 2012-05-19 10:04

\version "2.15.37"
\language "english"
\include "/Users/trevorbaca/Documents/abjad/trunk/abjad/cfg/abjad.scm"

#(set-global-staff-size 18)

\layout {
```

```
        indent = #0
        ragged-right = ##t
        \context {
            \Score
            \remove Bar_number_engraver
            \remove Default_bar_line_engraver
            \override Clef #'transparent = ##t
            \override SpacingSpanner #'strict-grace-spacing = ##t
            \override SpacingSpanner #'strict-note-spacing = ##t
            \override SpacingSpanner #'uniform-stretching = ##t
            \override TextScript #'staff-padding = #4
            proportionalNotationDuration = #(ly:make-moment 1 32)
        }
    }

    \paper {
        bottom-margin = #10
        left-margin = #10
        line-width = #150
        system-system-spacing = #'(
            (basic-distance . 0) (minimum-distance . 0) (padding . 15) (stretchability . 0))
    }

    \score {
        \new Score <<
            \new Staff {
                c4
                d4
                e4
                f4
            }
        >>
    }
```

Return LilyPond file.

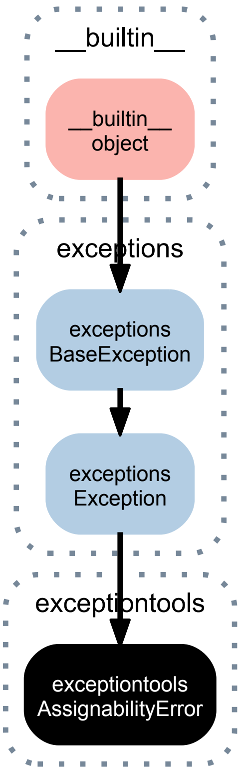
CHAPTER

FIFTYEIGHT

EXCEPTIONTOOLS

58.1 Concrete Classes

58.1.1 exceptiontools.AssignabilityError



class `tools.`**`AssignabilityError`**

Duration can not be assigned to note, rest or chord.

Special methods

`AssignabilityError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`AssignabilityError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`AssignabilityError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

AssignabilityError.**__repr__**() <==> *repr(x)*

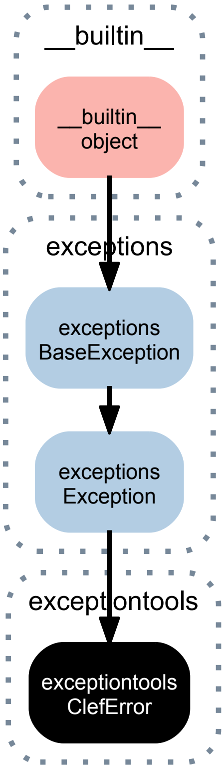
AssignabilityError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

AssignabilityError.**__setstate__**()

AssignabilityError.**__str__**() <==> *str(x)*

AssignabilityError.**__unicode__**()

58.1.2 exceptiontools.ClefError



class tools.**ClefError**
General clef error.

Special methods

ClefError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ClefError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ClefError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ClefError.**__repr__**() <==> *repr(x)*

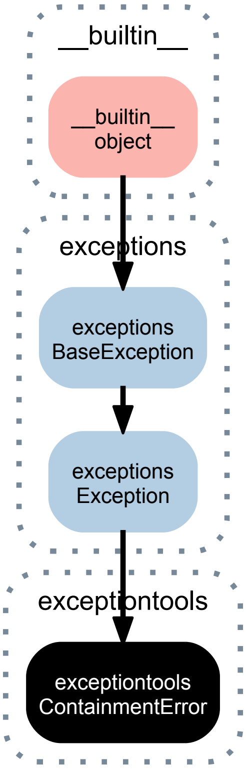
ClefError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ClefError.**__setstate__**()

ClefError.**__str__**() <==> *str(x)*

ClefError.**__unicode__**()

58.1.3 exceptiontools.ContainmentError



class tools.**ContainmentError**
General containment error.

Special methods

ContainmentError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ContainmentError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ContainmentError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ContainmentError.**__repr__**() <==> *repr(x)*

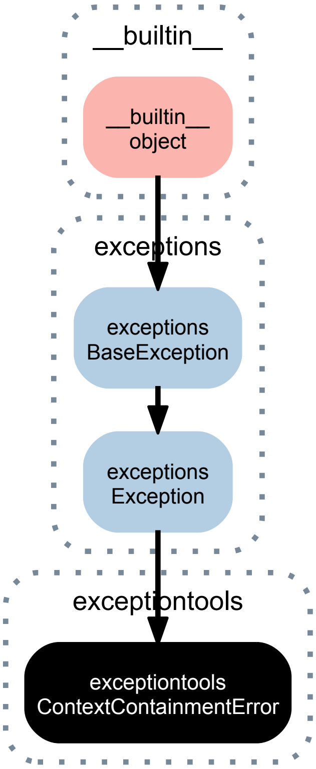
ContainmentError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ContainmentError.**__setstate__**()

ContainmentError.**__str__**() <==> *str(x)*

ContainmentError.**__unicode__**()

58.1.4 exceptiontools.ContextContainmentError



class tools.**ContextContainmentError**
Context can not contain other context.

Special methods

ContextContainmentError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ContextContainmentError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ContextContainmentError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ContextContainmentError.**__repr__**() <==> *repr(x)*

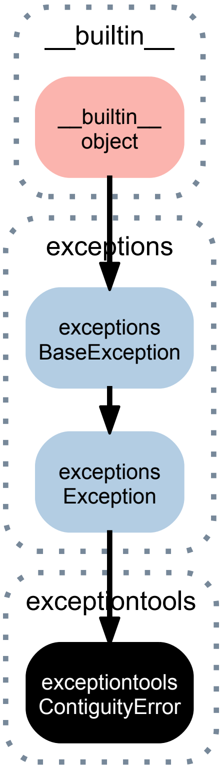
ContextContainmentError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ContextContainmentError.**__setstate__**()

ContextContainmentError.**__str__**() <==> *str(x)*

ContextContainmentError.**__unicode__**()

58.1.5 exceptiontools.ContiguityError



class `tools.`**`ContiguityError`**

Input is not contiguous.

Special methods

`ContiguityError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`ContiguityError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`ContiguityError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ContiguityError.`**`__repr__`**`()` <==> *repr(x)*

`ContiguityError.`**`__setattr__`**`()`

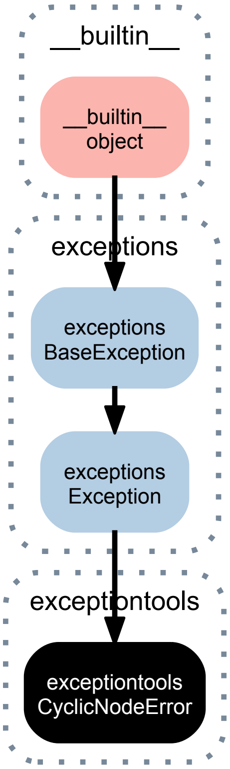
x.__setattr__('name', value) <==> x.name = value

`ContiguityError.`**`__setstate__`**`()`

`ContiguityError.`**`__str__`**`()` <==> *str(x)*

`ContiguityError.`**`__unicode__`**`()`

58.1.6 exceptiontools.CyclicNodeError



class tools.**CyclicNodeError**
Node is in cyclic relationship.

Special methods

CyclicNodeError.**__delattr__**()
x.__delattr__('name') <==> del x.name

CyclicNodeError.**__getitem__**()
x.__getitem__(y) <==> x[y]

CyclicNodeError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

CyclicNodeError.**__repr__**() <==> *repr(x)*

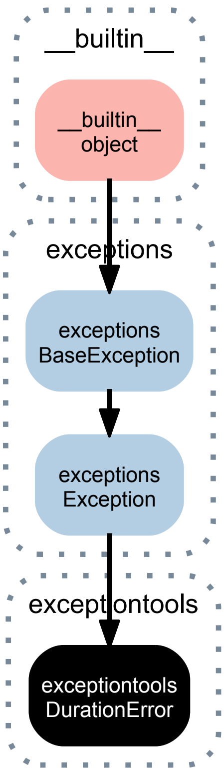
CyclicNodeError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

CyclicNodeError.**__setstate__**()

CyclicNodeError.**__str__**() <==> *str(x)*

CyclicNodeError.**__unicode__**()

58.1.7 exceptiontools.DurationError



class tools.**DurationError**
General duration error.

Special methods

DurationError.**__delattr__**()
x.__delattr__('name') <==> del x.name

DurationError.**__getitem__**()
x.__getitem__(y) <==> x[y]

DurationError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

DurationError.**__repr__**() <==> *repr(x)*

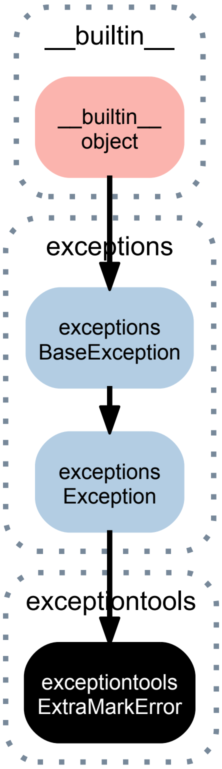
DurationError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

DurationError.**__setstate__**()

DurationError.**__str__**() <==> *str(x)*

DurationError.**__unicode__**()

58.1.8 exceptiontools.ExtraMarkError



class tools.**ExtraMarkError**
More than one mark is found for single-mark operation.

Special methods

ExtraMarkError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ExtraMarkError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ExtraMarkError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ExtraMarkError.**__repr__**() <==> *repr(x)*

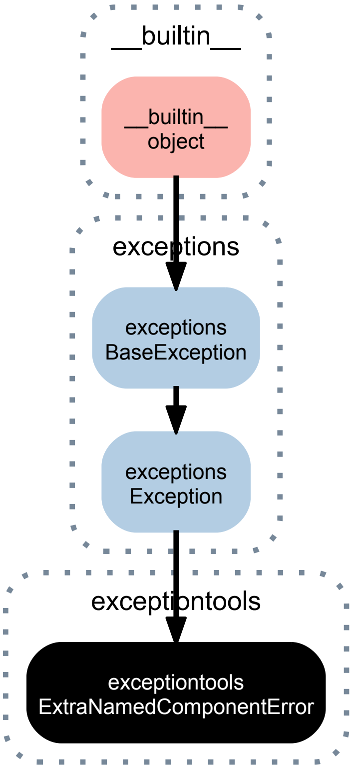
ExtraMarkError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ExtraMarkError.**__setstate__**()

ExtraMarkError.**__str__**() <==> *str(x)*

ExtraMarkError.**__unicode__**()

58.1.9 exceptiontools.ExtraNamedComponentError



class tools.**ExtraNamedComponentError**

More than one component with the same name is found..

Special methods

ExtraNamedComponentError.**__delattr__**()

x.__delattr__('name') <==> del x.name

ExtraNamedComponentError.**__getitem__**()

x.__getitem__(y) <==> x[y]

ExtraNamedComponentError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ExtraNamedComponentError.**__repr__**() <==> *repr(x)*

ExtraNamedComponentError.**__setattr__**()

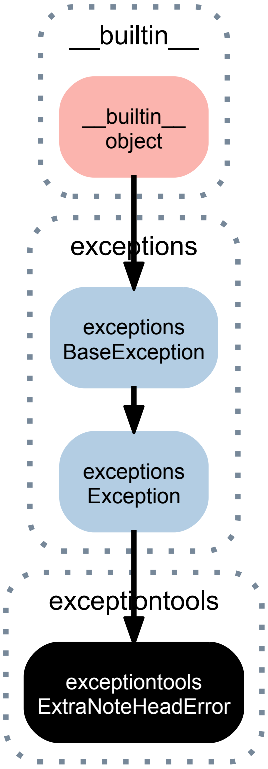
x.__setattr__('name', value) <==> x.name = value

ExtraNamedComponentError.**__setstate__**()

ExtraNamedComponentError.**__str__**() <==> *str(x)*

ExtraNamedComponentError.**__unicode__**()

58.1.10 exceptiontools.ExtraNoteHeadError



class `tools.`**`ExtraNoteHeadError`**
More than one note head found for single-note head operation.

Special methods

`ExtraNoteHeadError.`**`__delattr__`**`()`
x.__delattr__('name') <==> del x.name

`ExtraNoteHeadError.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`ExtraNoteHeadError.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ExtraNoteHeadError.`**`__repr__`**`()` <==> *repr(x)*

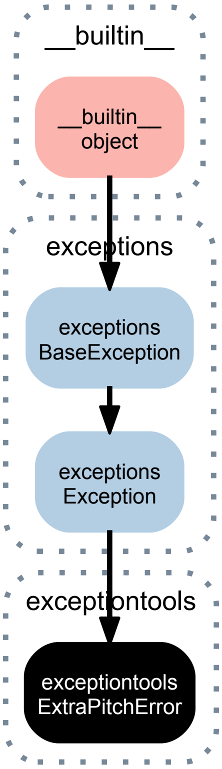
`ExtraNoteHeadError.`**`__setattr__`**`()`
x.__setattr__('name', value) <==> x.name = value

`ExtraNoteHeadError.`**`__setstate__`**`()`

`ExtraNoteHeadError.`**`__str__`**`()` <==> *str(x)*

`ExtraNoteHeadError.`**`__unicode__`**`()`

58.1.11 exceptiontools.ExtraPitchError



class `tools.`**`ExtraPitchError`**

More than one pitch found for single-pitch operation.

Special methods

`ExtraPitchError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`ExtraPitchError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`ExtraPitchError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ExtraPitchError.`**`__repr__`**`()` *<==> repr(x)*

`ExtraPitchError.`**`__setattr__`**`()`

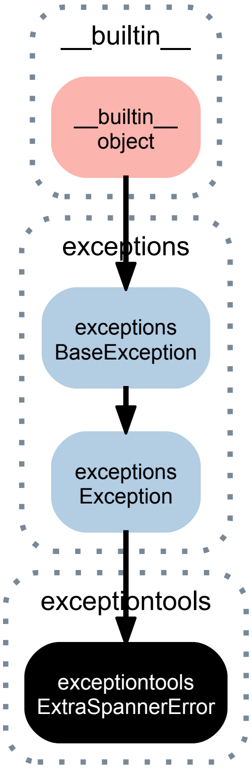
x.__setattr__('name', value) <==> x.name = value

`ExtraPitchError.`**`__setstate__`**`()`

`ExtraPitchError.`**`__str__`**`()` *<==> str(x)*

`ExtraPitchError.`**`__unicode__`**`()`

58.1.12 exceptiontools.ExtraSpannerError



class tools.**ExtraSpannerError**
More than one spanner found for single-spanner operation.

Special methods

ExtraSpannerError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ExtraSpannerError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ExtraSpannerError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ExtraSpannerError.**__repr__**() <==> *repr(x)*

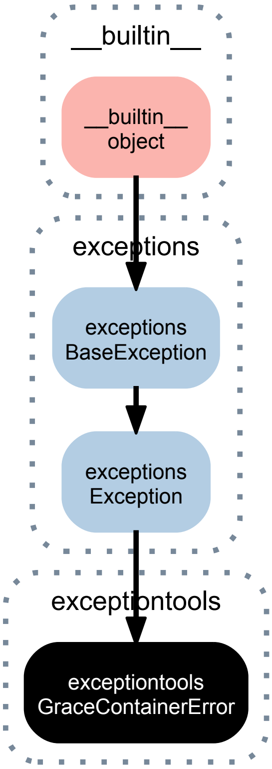
ExtraSpannerError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ExtraSpannerError.**__setstate__**()

ExtraSpannerError.**__str__**() <==> *str(x)*

ExtraSpannerError.**__unicode__**()

58.1.13 exceptiontools.GraceContainerError



class `tools.`**`GraceContainerError`**
General grace container error.

Special methods

`GraceContainerError.`**`__delattr__`**`()`
x.__delattr__('name') <==> del x.name

`GraceContainerError.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`GraceContainerError.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`GraceContainerError.`**`__repr__`**`()` <==> *repr(x)*

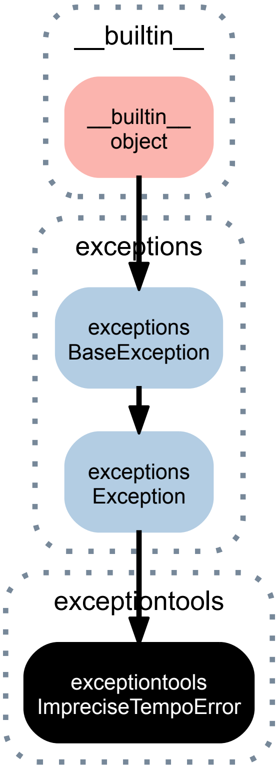
`GraceContainerError.`**`__setattr__`**`()`
x.__setattr__('name', value) <==> x.name = value

`GraceContainerError.`**`__setstate__`**`()`

`GraceContainerError.`**`__str__`**`()` <==> *str(x)*

`GraceContainerError.`**`__unicode__`**`()`

58.1.14 exceptiontools.ImpreciseTempoError



class tools.**ImpreciseTempoError**
TempoMark is imprecise.

Special methods

ImpreciseTempoError.**__delattr__**()
x.__delattr__('name') <==> del x.name

ImpreciseTempoError.**__getitem__**()
x.__getitem__(y) <==> x[y]

ImpreciseTempoError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

ImpreciseTempoError.**__repr__**() <==> *repr(x)*

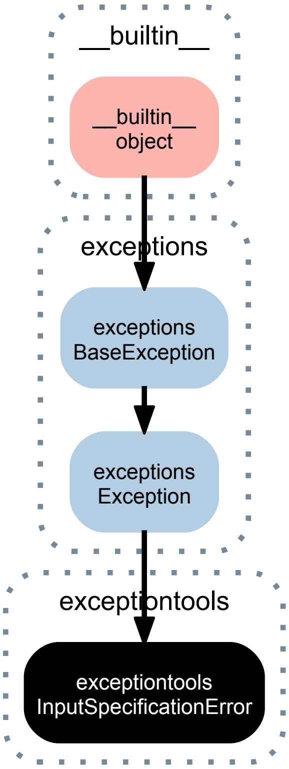
ImpreciseTempoError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

ImpreciseTempoError.**__setstate__**()

ImpreciseTempoError.**__str__**() <==> *str(x)*

ImpreciseTempoError.**__unicode__**()

58.1.15 exceptiontools.InputSpecificationError



class tools.**InputSpecificationError**
Input is badly formed.

Special methods

InputSpecificationError.**__delattr__**()
x.__delattr__('name') <==> del x.name

InputSpecificationError.**__getitem__**()
x.__getitem__(y) <==> x[y]

InputSpecificationError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

InputSpecificationError.**__repr__**() <==> *repr(x)*

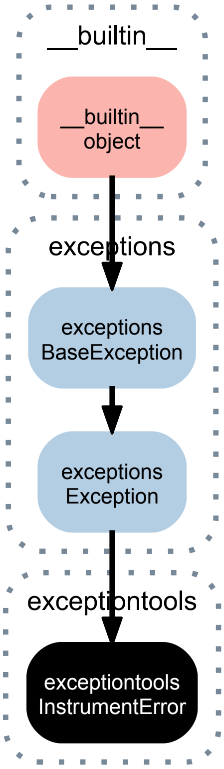
InputSpecificationError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

InputSpecificationError.**__setstate__**()

InputSpecificationError.**__str__**() <==> *str(x)*

InputSpecificationError.**__unicode__**()

58.1.16 exceptiontools.InstrumentError



class tools.**InstrumentError**

General instrument error.

Special methods

InstrumentError.**__delattr__**()

x.__delattr__('name') <==> del x.name

InstrumentError.**__getitem__**()

x.__getitem__(y) <==> x[y]

InstrumentError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

InstrumentError.**__repr__**() <==> *repr(x)*

InstrumentError.**__setattr__**()

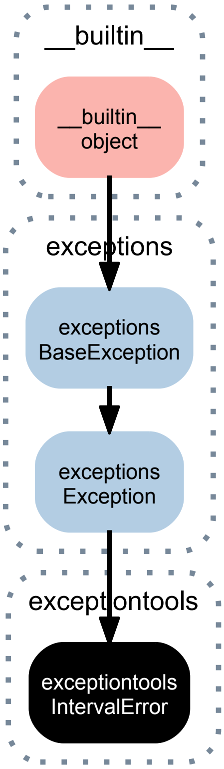
x.__setattr__('name', value) <==> x.name = value

InstrumentError.**__setstate__**()

InstrumentError.**__str__**() <==> *str(x)*

InstrumentError.**__unicode__**()

58.1.17 exceptiontools.IntervalError



class `tools.`**`IntervalError`**

General pitch interval error.

Special methods

`IntervalError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`IntervalError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`IntervalError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`IntervalError.`**`__repr__`**`()` <==> *repr(x)*

`IntervalError.`**`__setattr__`**`()`

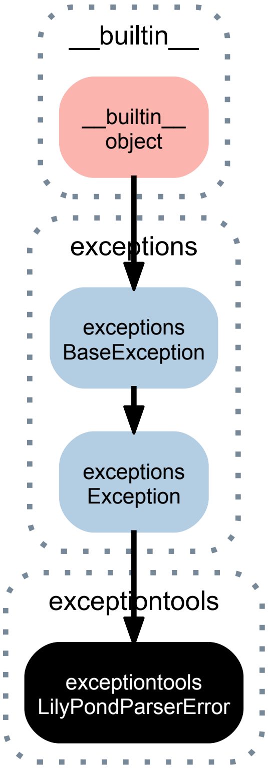
x.__setattr__('name', value) <==> x.name = value

`IntervalError.`**`__setstate__`**`()`

`IntervalError.`**`__str__`**`()` <==> *str(x)*

`IntervalError.`**`__unicode__`**`()`

58.1.18 exceptiontools.LilyPondParserError



class `tools.`**`LilyPondParserError`**

Can not parse input.

Special methods

`LilyPondParserError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`LilyPondParserError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`LilyPondParserError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`LilyPondParserError.`**`__repr__`**`()` <==> *repr(x)*

`LilyPondParserError.`**`__setattr__`**`()`

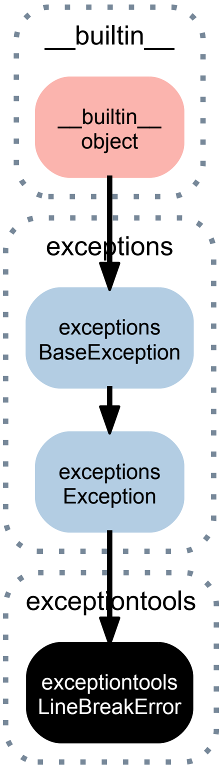
x.__setattr__('name', value) <==> x.name = value

`LilyPondParserError.`**`__setstate__`**`()`

`LilyPondParserError.`**`__str__`**`()` <==> *str(x)*

`LilyPondParserError.`**`__unicode__`**`()`

58.1.19 exceptiontools.LineBreakError



class tools.**LineBreakError**
General link break error.

Special methods

LineBreakError.**__delattr__**()
x.__delattr__('name') <==> del x.name

LineBreakError.**__getitem__**()
x.__getitem__(y) <==> x[y]

LineBreakError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

LineBreakError.**__repr__**() <==> *repr(x)*

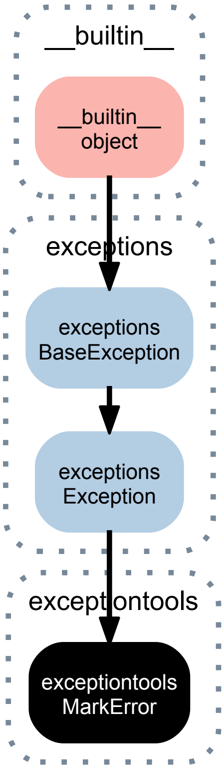
LineBreakError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

LineBreakError.**__setstate__**()

LineBreakError.**__str__**() <==> *str(x)*

LineBreakError.**__unicode__**()

58.1.20 exceptiontools.MarkError



class `tools.`**`MarkError`**
General mark error.

Special methods

`MarkError.`**`__delattr__`**`()`
x.__delattr__('name') <==> del x.name

`MarkError.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`MarkError.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MarkError.`**`__repr__`**`()` <==> *repr(x)*

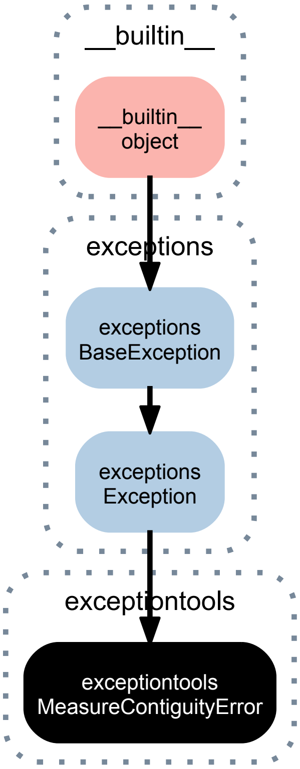
`MarkError.`**`__setattr__`**`()`
x.__setattr__('name', value) <==> x.name = value

`MarkError.`**`__setstate__`**`()`

`MarkError.`**`__str__`**`()` <==> *str(x)*

`MarkError.`**`__unicode__`**`()`

58.1.21 exceptiontools.MeasureContiguityError



class `tools.`**`MeasureContiguityError`**

Measures must be contiguous.

Special methods

`MeasureContiguityError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MeasureContiguityError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MeasureContiguityError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MeasureContiguityError.`**`__repr__`**`()` <==> *repr(x)*

`MeasureContiguityError.`**`__setattr__`**`()`

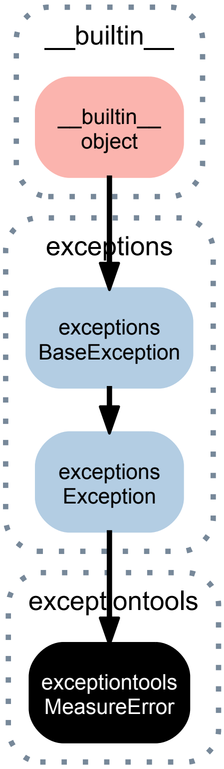
x.__setattr__('name', value) <==> x.name = value

`MeasureContiguityError.`**`__setstate__`**`()`

`MeasureContiguityError.`**`__str__`**`()` <==> *str(x)*

`MeasureContiguityError.`**`__unicode__`**`()`

58.1.22 exceptiontools.MeasureError



class `tools.`**`MeasureError`**

General measure error.

Special methods

`MeasureError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MeasureError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MeasureError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MeasureError.`**`__repr__`**`()` <==> *repr(x)*

`MeasureError.`**`__setattr__`**`()`

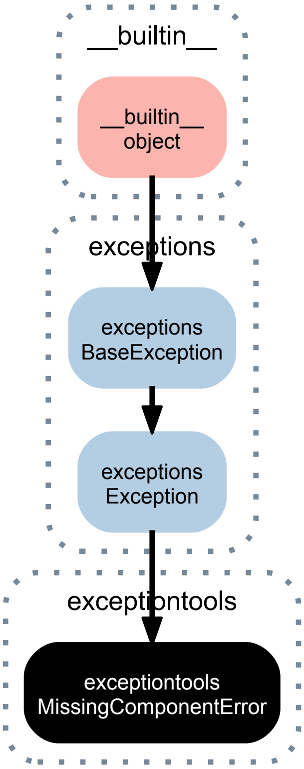
x.__setattr__('name', value) <==> x.name = value

`MeasureError.`**`__setstate__`**`()`

`MeasureError.`**`__str__`**`()` <==> *str(x)*

`MeasureError.`**`__unicode__`**`()`

58.1.23 exceptiontools.MissingComponentError



class `tools.`**`MissingComponentError`**

No component found.

Special methods

`MissingComponentError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingComponentError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingComponentError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingComponentError.`**`__repr__`**`()` <==> *repr(x)*

`MissingComponentError.`**`__setattr__`**`()`

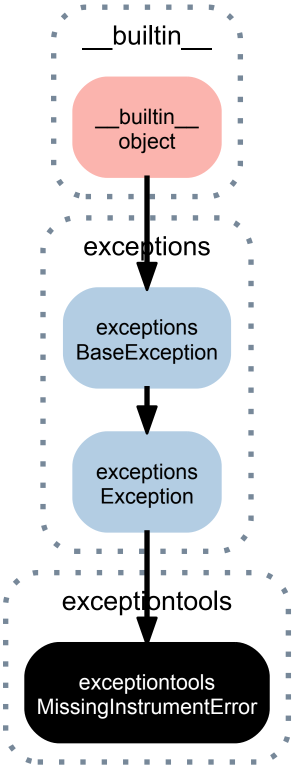
x.__setattr__('name', value) <==> x.name = value

`MissingComponentError.`**`__setstate__`**`()`

`MissingComponentError.`**`__str__`**`()` <==> *str(x)*

`MissingComponentError.`**`__unicode__`**`()`

58.1.24 exceptiontools.MissingInstrumentError



class tools.**MissingInstrumentError**
No instrument found.

Special methods

MissingInstrumentError.**__delattr__**()
x.__delattr__('name') <==> del x.name

MissingInstrumentError.**__getitem__**()
x.__getitem__(y) <==> x[y]

MissingInstrumentError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

MissingInstrumentError.**__repr__**() <==> *repr(x)*

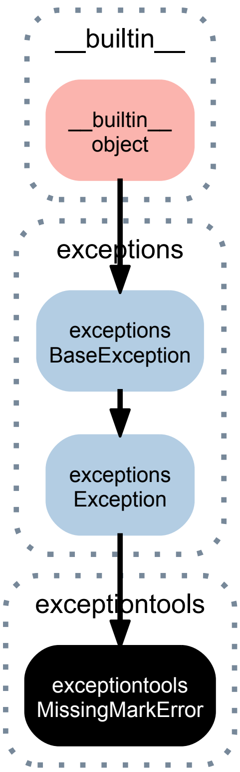
MissingInstrumentError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

MissingInstrumentError.**__setstate__**()

MissingInstrumentError.**__str__**() <==> *str(x)*

MissingInstrumentError.**__unicode__**()

58.1.25 exceptiontools.MissingMarkError



class `tools.`**`MissingMarkError`**

No mark found.

Special methods

`MissingMarkError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingMarkError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingMarkError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingMarkError.`**`__repr__`**() <==> *repr(x)*

`MissingMarkError.`**`__setattr__`**`()`

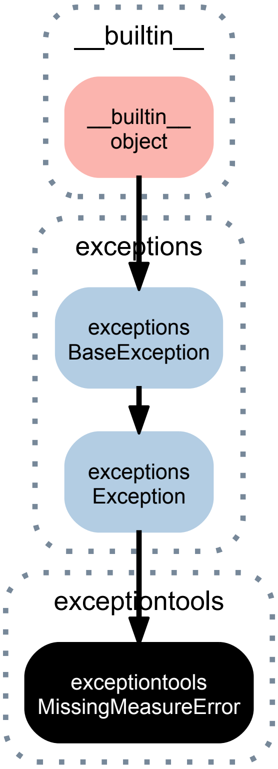
x.__setattr__('name', value) <==> x.name = value

`MissingMarkError.`**`__setstate__`**`()`

`MissingMarkError.`**`__str__`**() <==> *str(x)*

`MissingMarkError.`**`__unicode__`**`()`

58.1.26 exceptiontools.MissingMeasureError



class `tools.`**`MissingMeasureError`**
No measure found.

Special methods

`MissingMeasureError.`**`__delattr__`**`()`
x.__delattr__('name') <==> del x.name

`MissingMeasureError.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`MissingMeasureError.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingMeasureError.`**`__repr__`**`()` <==> *repr(x)*

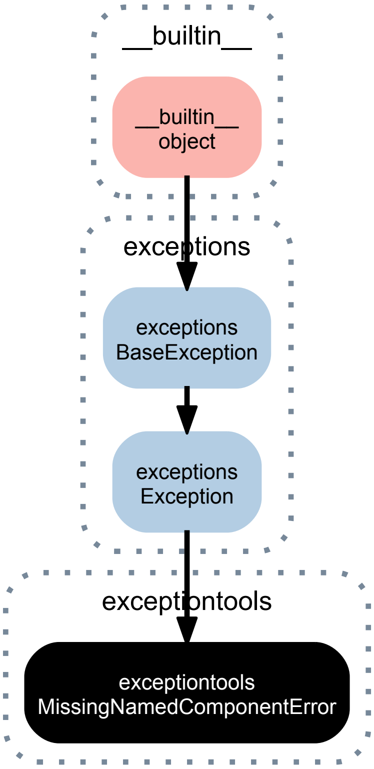
`MissingMeasureError.`**`__setattr__`**`()`
x.__setattr__('name', value) <==> x.name = value

`MissingMeasureError.`**`__setstate__`**`()`

`MissingMeasureError.`**`__str__`**`()` <==> *str(x)*

`MissingMeasureError.`**`__unicode__`**`()`

58.1.27 exceptiontools.MissingNamedComponentError



class `tools.`**`MissingNamedComponentError`**

No named component found.

Special methods

`MissingNamedComponentError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingNamedComponentError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingNamedComponentError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingNamedComponentError.`**`__repr__`**`()` <==> *repr(x)*

`MissingNamedComponentError.`**`__setattr__`**`()`

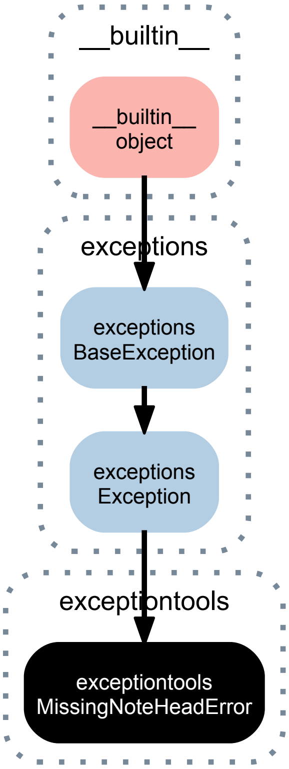
x.__setattr__('name', value) <==> x.name = value

`MissingNamedComponentError.`**`__setstate__`**`()`

`MissingNamedComponentError.`**`__str__`**`()` <==> *str(x)*

`MissingNamedComponentError.`**`__unicode__`**`()`

58.1.28 exceptiontools.MissingNoteHeadError



class `tools.`**`MissingNoteHeadError`**

No note head found.

Special methods

`MissingNoteHeadError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingNoteHeadError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingNoteHeadError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingNoteHeadError.`**`__repr__`**`()` <==> *repr(x)*

`MissingNoteHeadError.`**`__setattr__`**`()`

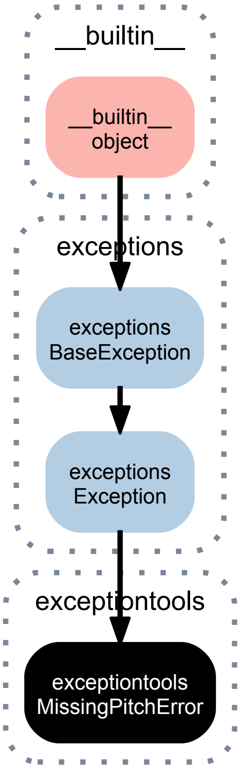
x.__setattr__('name', value) <==> x.name = value

`MissingNoteHeadError.`**`__setstate__`**`()`

`MissingNoteHeadError.`**`__str__`**`()` <==> *str(x)*

`MissingNoteHeadError.`**`__unicode__`**`()`

58.1.29 exceptiontools.MissingPitchError



class `tools.`**`MissingPitchError`**

No pitch found.

Special methods

`MissingPitchError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingPitchError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingPitchError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingPitchError.`**`__repr__`**`()` <==> *repr(x)*

`MissingPitchError.`**`__setattr__`**`()`

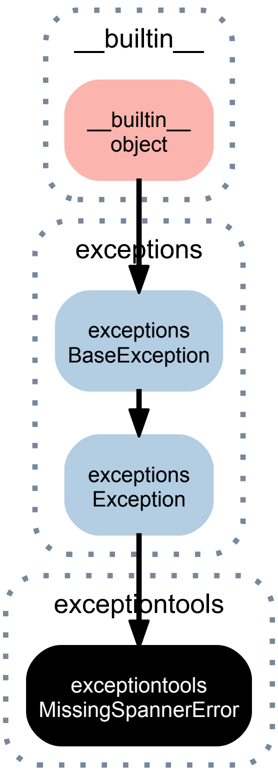
x.__setattr__('name', value) <==> x.name = value

`MissingPitchError.`**`__setstate__`**`()`

`MissingPitchError.`**`__str__`**`()` <==> *str(x)*

`MissingPitchError.`**`__unicode__`**`()`

58.1.30 exceptiontools.MissingSpannerError



class tools.**MissingSpannerError**
No spanner found.

Special methods

MissingSpannerError.**__delattr__**()
x.__delattr__('name') <==> del x.name

MissingSpannerError.**__getitem__**()
x.__getitem__(y) <==> x[y]

MissingSpannerError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

MissingSpannerError.**__repr__**() <==> *repr(x)*

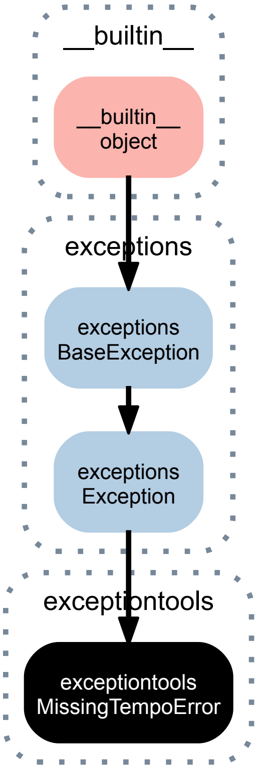
MissingSpannerError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

MissingSpannerError.**__setstate__**()

MissingSpannerError.**__str__**() <==> *str(x)*

MissingSpannerError.**__unicode__**()

58.1.31 exceptiontools.MissingTempoError



class `tools.`**`MissingTempoError`**

No tempo found.

Special methods

`MissingTempoError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MissingTempoError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MissingTempoError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MissingTempoError.`**`__repr__`**`()` <==> *repr(x)*

`MissingTempoError.`**`__setattr__`**`()`

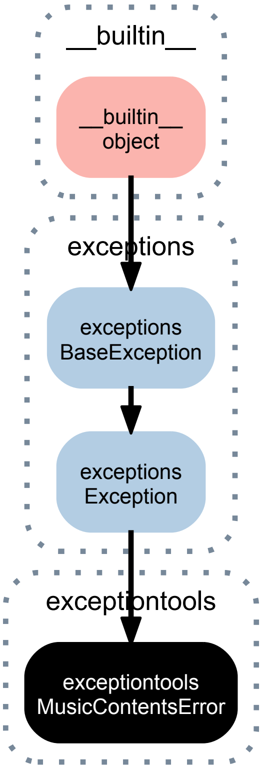
x.__setattr__('name', value) <==> x.name = value

`MissingTempoError.`**`__setstate__`**`()`

`MissingTempoError.`**`__str__`**`()` <==> *str(x)*

`MissingTempoError.`**`__unicode__`**`()`

58.1.32 exceptiontools.MusicContentsError



class `tools.`**`MusicContentsError`**

General container contents error.

Special methods

`MusicContentsError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`MusicContentsError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`MusicContentsError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`MusicContentsError.`**`__repr__`**`()` <==> *repr(x)*

`MusicContentsError.`**`__setattr__`**`()`

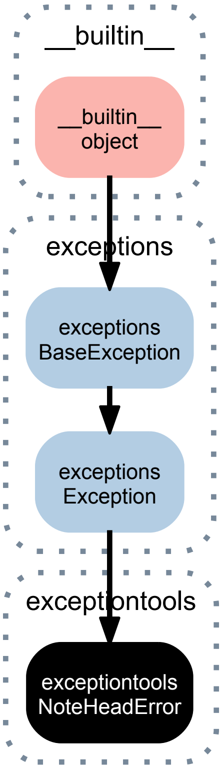
x.__setattr__('name', value) <==> x.name = value

`MusicContentsError.`**`__setstate__`**`()`

`MusicContentsError.`**`__str__`**`()` <==> *str(x)*

`MusicContentsError.`**`__unicode__`**`()`

58.1.33 exceptiontools.NoteHeadError



class tools.**NoteHeadError**
General note head error.

Special methods

NoteHeadError.**__delattr__**()
x.__delattr__('name') <==> del x.name

NoteHeadError.**__getitem__**()
x.__getitem__(y) <==> x[y]

NoteHeadError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

NoteHeadError.**__repr__**() <==> *repr(x)*

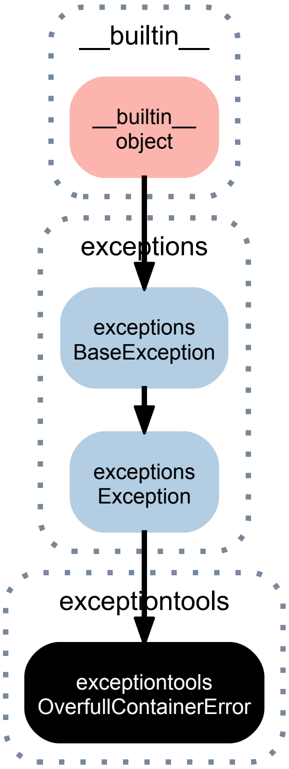
NoteHeadError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

NoteHeadError.**__setstate__**()

NoteHeadError.**__str__**() <==> *str(x)*

NoteHeadError.**__unicode__**()

58.1.34 exceptiontools.OverfullContainerError



class `tools.`**`OverfullContainerError`**

Container contents duration is greater than container target duration.

Special methods

`OverfullContainerError.`**`__delattr__`**()

x.__delattr__('name') <==> del x.name

`OverfullContainerError.`**`__getitem__`**()

x.__getitem__(y) <==> x[y]

`OverfullContainerError.`**`__getslice__`**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`OverfullContainerError.`**`__repr__`**() <==> *repr(x)*

`OverfullContainerError.`**`__setattr__`**()

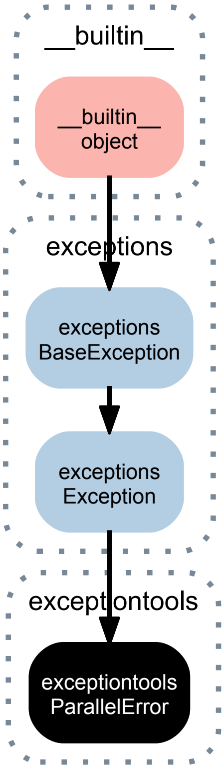
x.__setattr__('name', value) <==> x.name = value

`OverfullContainerError.`**`__setstate__`**()

`OverfullContainerError.`**`__str__`**() <==> *str(x)*

`OverfullContainerError.`**`__unicode__`**()

58.1.35 exceptiontools.ParallelError



class `tools.`**`ParallelError`**

Parallel containers must contain contexts only.

Special methods

`ParallelError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`ParallelError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`ParallelError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`ParallelError.`**`__repr__`**`()` <==> *repr(x)*

`ParallelError.`**`__setattr__`**`()`

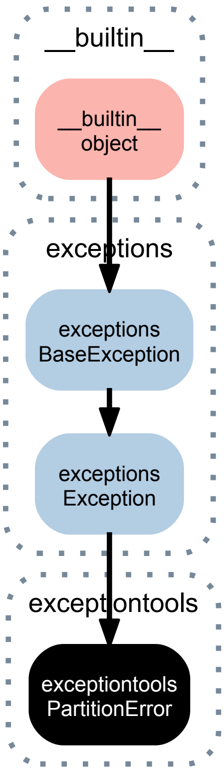
x.__setattr__('name', value) <==> x.name = value

`ParallelError.`**`__setstate__`**`()`

`ParallelError.`**`__str__`**`()` <==> *str(x)*

`ParallelError.`**`__unicode__`**`()`

58.1.36 exceptiontools.PartitionError



class `tools.`**`PartitionError`**

General partition error.

Special methods

`PartitionError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`PartitionError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`PartitionError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`PartitionError.`**`__repr__`**`()` <==> *repr(x)*

`PartitionError.`**`__setattr__`**`()`

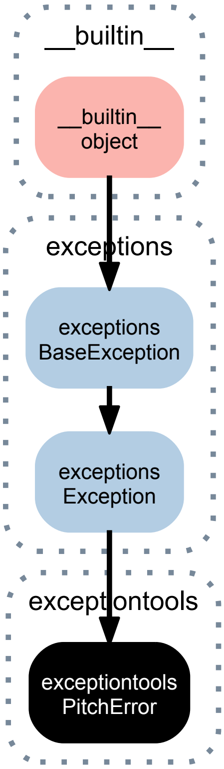
x.__setattr__('name', value) <==> x.name = value

`PartitionError.`**`__setstate__`**`()`

`PartitionError.`**`__str__`**`()` <==> *str(x)*

`PartitionError.`**`__unicode__`**`()`

58.1.37 exceptiontools.PitchError



class `tools.`**`PitchError`**

General pitch error.

Special methods

`PitchError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`PitchError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`PitchError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`PitchError.`**`__repr__`**`()` <==> *repr(x)*

`PitchError.`**`__setattr__`**`()`

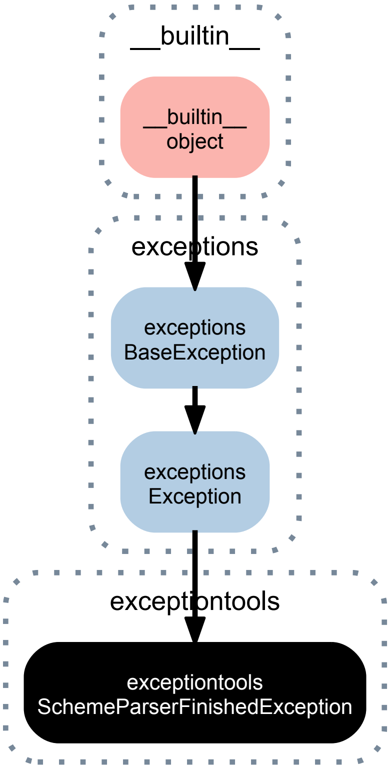
x.__setattr__('name', value) <==> x.name = value

`PitchError.`**`__setstate__`**`()`

`PitchError.`**`__str__`**`()` <==> *str(x)*

`PitchError.`**`__unicode__`**`()`

58.1.38 exceptiontools.SchemeParserFinishedException



class tools.**SchemeParserFinishedException**
SchemeParser has finished parsing.

Special methods

SchemeParserFinishedException.**__delattr__**()
x.__delattr__('name') <==> del x.name

SchemeParserFinishedException.**__getitem__**()
x.__getitem__(y) <==> x[y]

SchemeParserFinishedException.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

SchemeParserFinishedException.**__repr__**() <==> *repr(x)*

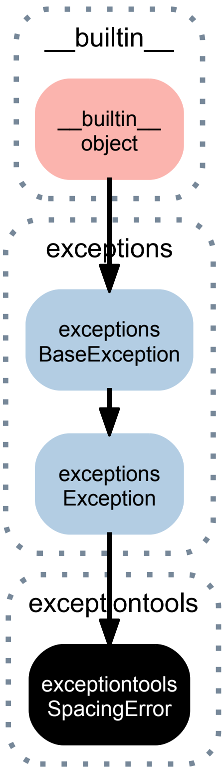
SchemeParserFinishedException.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

SchemeParserFinishedException.**__setstate__**()

SchemeParserFinishedException.**__str__**() <==> *str(x)*

SchemeParserFinishedException.**__unicode__**()

58.1.39 exceptiontools.SpacingError



class `tools.`**`SpacingError`**

General spacing error.

Special methods

`SpacingError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`SpacingError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`SpacingError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`SpacingError.`**`__repr__`**`()` <==> *repr(x)*

`SpacingError.`**`__setattr__`**`()`

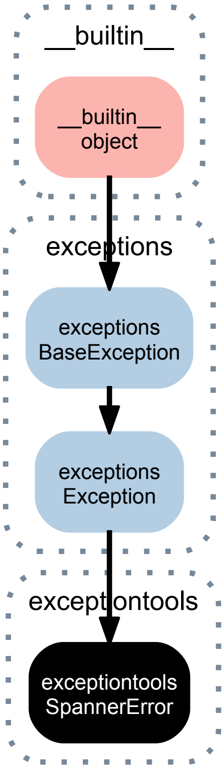
x.__setattr__('name', value) <==> x.name = value

`SpacingError.`**`__setstate__`**`()`

`SpacingError.`**`__str__`**`()` <==> *str(x)*

`SpacingError.`**`__unicode__`**`()`

58.1.40 exceptiontools.SpannerError



class tools.**SpannerError**
General spanner error.

Special methods

SpannerError.**__delattr__**()
x.__delattr__('name') <==> del x.name

SpannerError.**__getitem__**()
x.__getitem__(y) <==> x[y]

SpannerError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

SpannerError.**__repr__**() <==> *repr(x)*

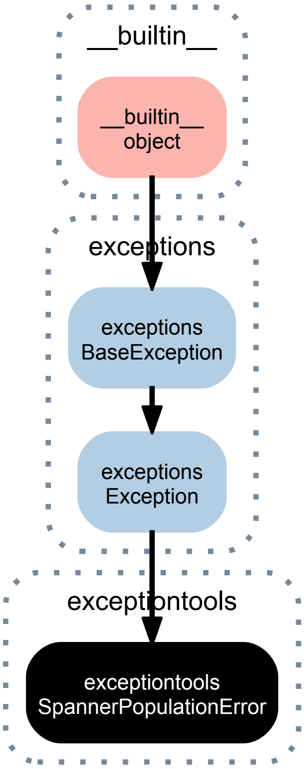
SpannerError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

SpannerError.**__setstate__**()

SpannerError.**__str__**() <==> *str(x)*

SpannerError.**__unicode__**()

58.1.41 exceptiontools.SpannerPopulationError



class `tools.`**`SpannerPopulationError`**

Spanner contents incorrect.

Spanner may be missing component it is assumed to have.

Spanner may have a component it is assumed not to have.

Special methods

`SpannerPopulationError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`SpannerPopulationError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`SpannerPopulationError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`SpannerPopulationError.`**`__repr__`**`()` *<==> repr(x)*

`SpannerPopulationError.`**`__setattr__`**`()`

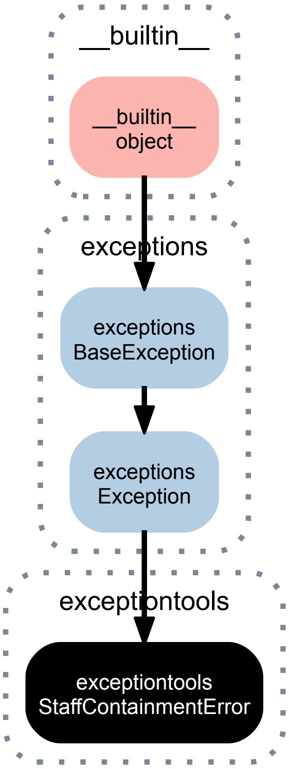
x.__setattr__('name', value) <==> x.name = value

`SpannerPopulationError.`**`__setstate__`**`()`

`SpannerPopulationError.`**`__str__`**`()` *<==> str(x)*

`SpannerPopulationError.`**`__unicode__`**`()`

58.1.42 exceptiontools.StaffContainmentError



class tools.**StaffContainmentError**

Staves must not contain staff groups, scores or other staves.

Special methods

StaffContainmentError.**__delattr__**()

x.__delattr__('name') <==> del x.name

StaffContainmentError.**__getitem__**()

x.__getitem__(y) <==> x[y]

StaffContainmentError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

StaffContainmentError.**__repr__**() <==> *repr(x)*

StaffContainmentError.**__setattr__**()

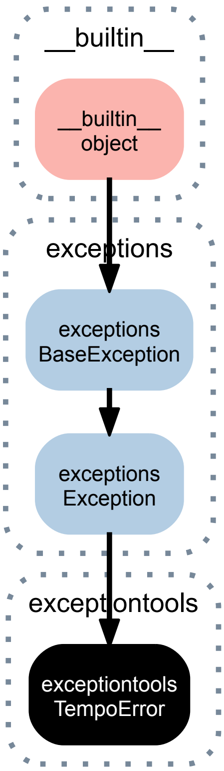
x.__setattr__('name', value) <==> x.name = value

StaffContainmentError.**__setstate__**()

StaffContainmentError.**__str__**() <==> *str(x)*

StaffContainmentError.**__unicode__**()

58.1.43 exceptiontools.TempoError



class tools.**TempoError**
General tempo error.

Special methods

TempoError.**__delattr__**()
x.__delattr__('name') <==> del x.name

TempoError.**__getitem__**()
x.__getitem__(y) <==> x[y]

TempoError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TempoError.**__repr__**() <==> *repr(x)*

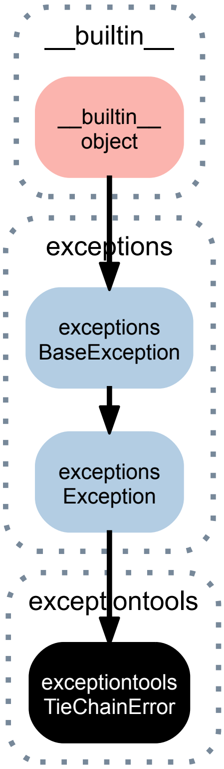
TempoError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

TempoError.**__setstate__**()

TempoError.**__str__**() <==> *str(x)*

TempoError.**__unicode__**()

58.1.44 exceptiontools.TieChainError



class `tools.`**`TieChainError`**

General tie chain error.

Special methods

`TieChainError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`TieChainError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`TieChainError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`TieChainError.`**`__repr__`**`()` <==> *repr(x)*

`TieChainError.`**`__setattr__`**`()`

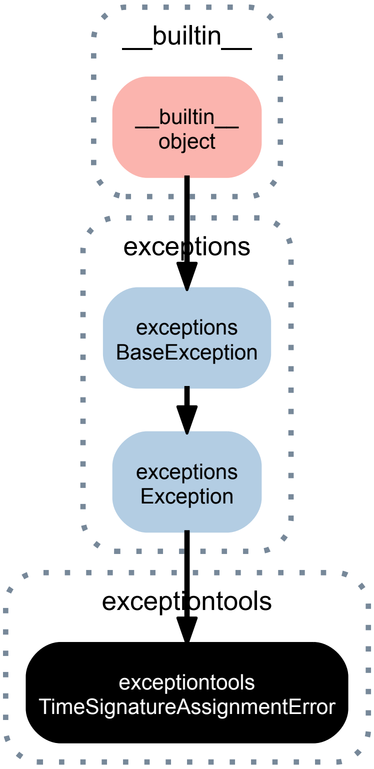
x.__setattr__('name', value) <==> x.name = value

`TieChainError.`**`__setstate__`**`()`

`TieChainError.`**`__str__`**`()` <==> *str(x)*

`TieChainError.`**`__unicode__`**`()`

58.1.45 exceptiontools.TimeSignatureAssignmentError



class `tools.`**`TimeSignatureAssignmentError`**
Can not assign time signature to dynamic measure.

Special methods

`TimeSignatureAssignmentError.`**`__delattr__`**`()`
x.__delattr__('name') <==> del x.name

`TimeSignatureAssignmentError.`**`__getitem__`**`()`
x.__getitem__(y) <==> x[y]

`TimeSignatureAssignmentError.`**`__getslice__`**`()`
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`TimeSignatureAssignmentError.`**`__repr__`**`()` <==> *repr(x)*

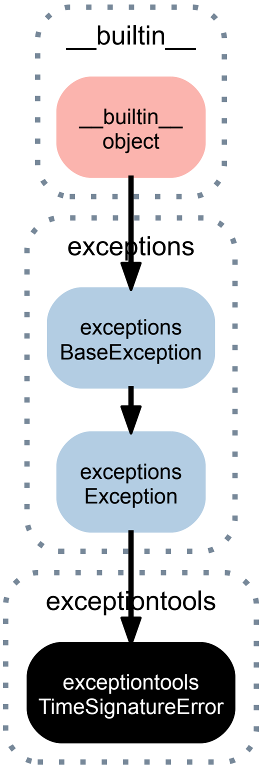
`TimeSignatureAssignmentError.`**`__setattr__`**`()`
x.__setattr__('name', value) <==> x.name = value

`TimeSignatureAssignmentError.`**`__setstate__`**`()`

`TimeSignatureAssignmentError.`**`__str__`**`()` <==> *str(x)*

`TimeSignatureAssignmentError.`**`__unicode__`**`()`

58.1.46 exceptiontools.TimeSignatureError



class tools.**TimeSignatureError**
General time signature error.

Special methods

TimeSignatureError.**__delattr__**()
x.__delattr__('name') <==> del x.name

TimeSignatureError.**__getitem__**()
x.__getitem__(y) <==> x[y]

TimeSignatureError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TimeSignatureError.**__repr__**() <==> *repr(x)*

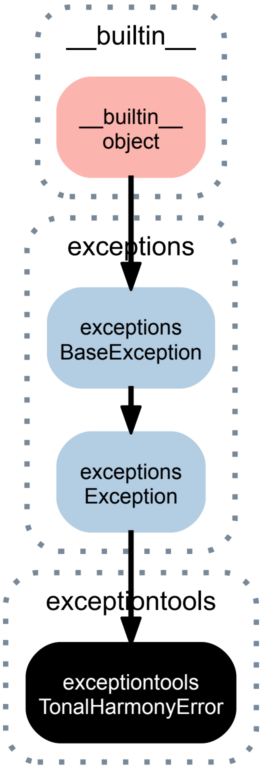
TimeSignatureError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

TimeSignatureError.**__setstate__**()

TimeSignatureError.**__str__**() <==> *str(x)*

TimeSignatureError.**__unicode__**()

58.1.47 exceptiontools.TonalHarmonyError



class `tools.`**`TonalHarmonyError`**

General tonal harmony error.

Special methods

`TonalHarmonyError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`TonalHarmonyError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`TonalHarmonyError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`TonalHarmonyError.`**`__repr__`**`()` <==> *repr(x)*

`TonalHarmonyError.`**`__setattr__`**`()`

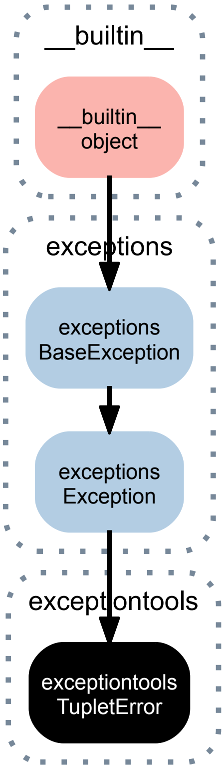
x.__setattr__('name', value) <==> x.name = value

`TonalHarmonyError.`**`__setstate__`**`()`

`TonalHarmonyError.`**`__str__`**`()` <==> *str(x)*

`TonalHarmonyError.`**`__unicode__`**`()`

58.1.48 exceptiontools.TupletError



class tools.**TupletError**
Geneal tuplet error.

Special methods

TupletError.**__delattr__**()
x.__delattr__('name') <==> del x.name

TupletError.**__getitem__**()
x.__getitem__(y) <==> x[y]

TupletError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TupletError.**__repr__**() <==> *repr(x)*

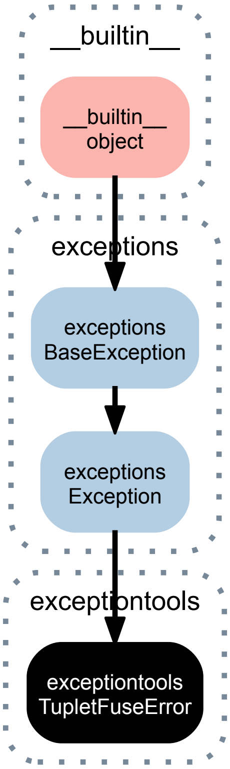
TupletError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

TupletError.**__setstate__**()

TupletError.**__str__**() <==> *str(x)*

TupletError.**__unicode__**()

58.1.49 exceptiontools.TupletFuseError



class tools.**TupletFuseError**

Tuplets must carry same multiplier and be same type in order to fuse correctly.

Special methods

TupletFuseError.**__delattr__**()

x.__delattr__('name') <==> del x.name

TupletFuseError.**__getitem__**()

x.__getitem__(y) <==> x[y]

TupletFuseError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TupletFuseError.**__repr__**() <==> *repr(x)*

TupletFuseError.**__setattr__**()

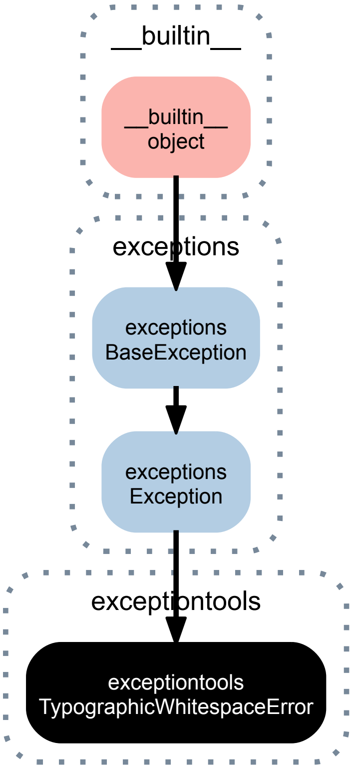
x.__setattr__('name', value) <==> x.name = value

TupletFuseError.**__setstate__**()

TupletFuseError.**__str__**() <==> *str(x)*

TupletFuseError.**__unicode__**()

58.1.50 exceptiontools.TypographicWhitespaceError



class tools.**TypographicWhitespaceError**
Whitespace after leaf confuses LilyPond timekeeping.

Special methods

TypographicWhitespaceError.**__delattr__**()
x.__delattr__('name') <==> del x.name

TypographicWhitespaceError.**__getitem__**()
x.__getitem__(y) <==> x[y]

TypographicWhitespaceError.**__getslice__**()
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

TypographicWhitespaceError.**__repr__**() <==> *repr(x)*

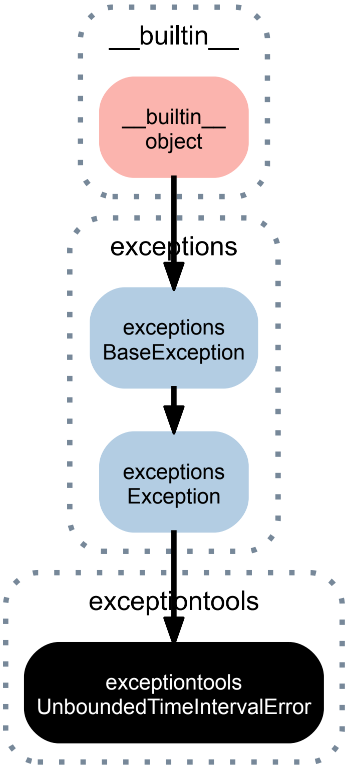
TypographicWhitespaceError.**__setattr__**()
x.__setattr__('name', value) <==> x.name = value

TypographicWhitespaceError.**__setstate__**()

TypographicWhitespaceError.**__str__**() <==> *str(x)*

TypographicWhitespaceError.**__unicode__**()

58.1.51 exceptiontools.UnboundedTimeIntervalError



class `tools.`**`UnboundedTimeIntervalError`**

Time interval has no bounds.

Special methods

`UnboundedTimeIntervalError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`UnboundedTimeIntervalError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`UnboundedTimeIntervalError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`UnboundedTimeIntervalError.`**`__repr__`**() <==> *repr(x)*

`UnboundedTimeIntervalError.`**`__setattr__`**`()`

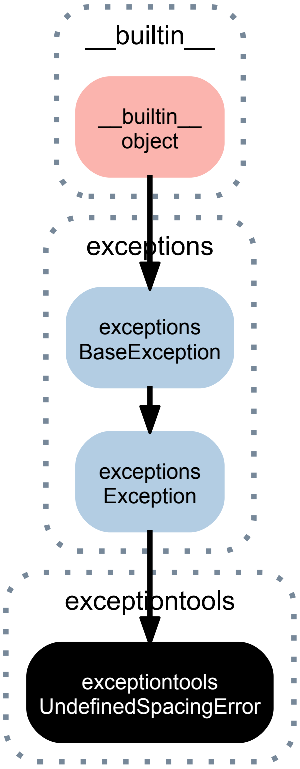
x.__setattr__('name', value) <==> x.name = value

`UnboundedTimeIntervalError.`**`__setstate__`**`()`

`UnboundedTimeIntervalError.`**`__str__`**() <==> *str(x)*

`UnboundedTimeIntervalError.`**`__unicode__`**`()`

58.1.52 exceptiontools.UndefinedSpacingError



class tools.**UndefinedSpacingError**

No spacing value found.

Special methods

UndefinedSpacingError.**__delattr__**()

x.__delattr__('name') <==> del x.name

UndefinedSpacingError.**__getitem__**()

x.__getitem__(y) <==> x[y]

UndefinedSpacingError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

UndefinedSpacingError.**__repr__**() <==> *repr(x)*

UndefinedSpacingError.**__setattr__**()

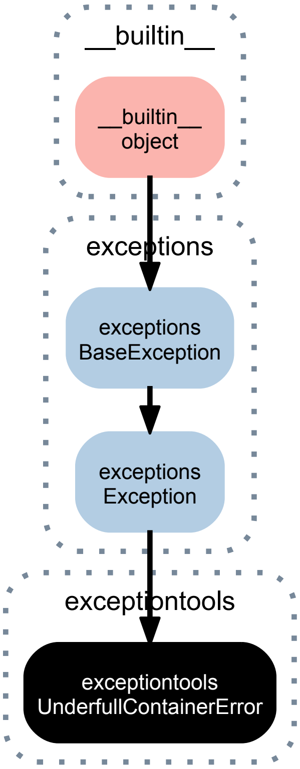
x.__setattr__('name', value) <==> x.name = value

UndefinedSpacingError.**__setstate__**()

UndefinedSpacingError.**__str__**() <==> *str(x)*

UndefinedSpacingError.**__unicode__**()

58.1.53 exceptiontools.UnderfullContainerError



class `tools.`**`UnderfullContainerError`**

Container contents duration is less than container target duration.

Special methods

`UnderfullContainerError.`**`__delattr__`**`()`

x.__delattr__('name') <==> del x.name

`UnderfullContainerError.`**`__getitem__`**`()`

x.__getitem__(y) <==> x[y]

`UnderfullContainerError.`**`__getslice__`**`()`

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

`UnderfullContainerError.`**`__repr__`**`()` <==> *repr(x)*

`UnderfullContainerError.`**`__setattr__`**`()`

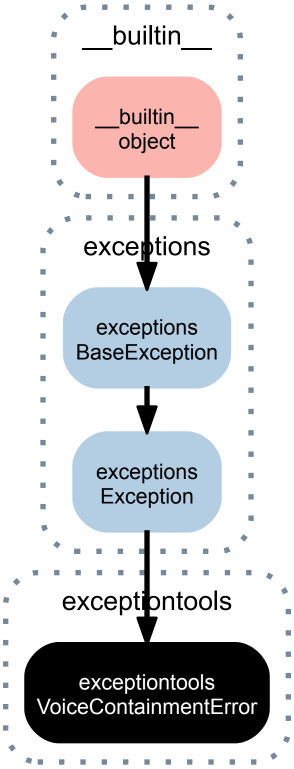
x.__setattr__('name', value) <==> x.name = value

`UnderfullContainerError.`**`__setstate__`**`()`

`UnderfullContainerError.`**`__str__`**`()` <==> *str(x)*

`UnderfullContainerError.`**`__unicode__`**`()`

58.1.54 exceptiontools.VoiceContainmentError



class tools.**VoiceContainmentError**

Voice must not contain staves, staff groups or scores.

Special methods

VoiceContainmentError.**__delattr__**()

x.__delattr__('name') <==> del x.name

VoiceContainmentError.**__getitem__**()

x.__getitem__(y) <==> x[y]

VoiceContainmentError.**__getslice__**()

x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

VoiceContainmentError.**__repr__**() <==> *repr(x)*

VoiceContainmentError.**__setattr__**()

x.__setattr__('name', value) <==> x.name = value

VoiceContainmentError.**__setstate__**()

VoiceContainmentError.**__str__**() <==> *str(x)*

VoiceContainmentError.**__unicode__**()

CHAPTER

FIFTYNINE

FORMATTOOLS

59.1 Functions

59.1.1 formattools.get_all_format_contributions

`formattools.get_all_format_contributions(component)`

Get all format contributions for *component*.

Return nested dictionary.

59.1.2 formattools.get_all_mark_format_contributions

`formattools.get_all_mark_format_contributions(component)`

Get all mark format contributions as nested dictionaries.

The first level of keys represent format slots.

The second level of keys represent format contributor ('articulations', 'markup', etc.).

Return dict.

59.1.3 formattools.get_articulation_format_contributions

`formattools.get_articulation_format_contributions(component)`

New in version 2.0. Get articulation format contributions for *component*.

Return list.

59.1.4 formattools.get_comment_format_contributions_for_slot

`formattools.get_comment_format_contributions_for_slot(component, slot)`

New in version 2.0. Get comment format contributions for *component* at *slot*.

Return list.

59.1.5 formattools.get_context_mark_format_contributions_for_slot

`formattools.get_context_mark_format_contributions_for_slot(component, slot)`

New in version 2.0. Get context mark format contributions for *component* at *slot*.

Return list.

59.1.6 formattools.get_context_mark_format_pieces

formattools.**get_context_mark_format_pieces**(*mark*)
Get context mark format pieces for *mark*.

Return list.

59.1.7 formattools.get_context_setting_format_contributions

formattools.**get_context_setting_format_contributions**(*component*)
New in version 2.0. Get context setting format contributions for *component*.

Return sorted list.

59.1.8 formattools.get_grob_override_format_contributions

formattools.**get_grob_override_format_contributions**(*component*)
New in version 2.0. Get grob override format contributions for *component*.

Return alphabetized list of LilyPond grob overrides.

59.1.9 formattools.get_grob_revert_format_contributions

formattools.**get_grob_revert_format_contributions**(*component*)
New in version 2.0. Get grob revert format contributions.

Return alphabetized list of LilyPond grob reverts.

59.1.10 formattools.get_lilypond_command_mark_format_contributions_for_slot

formattools.**get_lilypond_command_mark_format_contributions_for_slot**(*component*, *slot*)
New in version 2.0. Get LilyPond command mark format contributions for *component* at *slot*.

Return list.

59.1.11 formattools.get_markup_format_contributions

formattools.**get_markup_format_contributions**(*component*)
New in version 2.0. Get markup format contributions for *component*.

Return list.

59.1.12 formattools.get_spanner_format_contributions

formattools.**get_spanner_format_contributions**(*component*)
Get spanner format contributions for *component*.

Return dictionary with format slot keys and format contributions values.

59.1.13 formattools.get_stem_tremolo_format_contributions

formattools.**get_stem_tremolo_format_contributions**(*component*)
New in version 2.0. Get stem tremolo format contributions for *component*.

Return list.

59.1.14 formattools.is_formattable_context_mark_for_component

formattools.**is_formattable_context_mark_for_component**(*mark*, *component*)
Return True if ContextMark *mark* can format for *component*.

59.1.15 formattools.report_component_format_contributions

formattools.**report_component_format_contributions**(*component*, *verbose=False*)
New in version 1.1. Report *component* format contributions as string.

Set *verbose* to True or False.

59.1.16 formattools.report_spanner_format_contributions

formattools.**report_spanner_format_contributions**(*spanner*, *screen=True*)
New in version 2.9. Report spanner format contributions for every leaf to which spanner attaches.

```
>>> staff = Staff("c8 d e f")
>>> spanner = beamtools.BeamSpanner(staff[:])
```

```
>>> formattools.report_spanner_format_contributions(spanner)
c8  before: []
     after: []
     right: ['[']

d8  before: []
     after: []
     right: []

e8  before: []
     after: []
     right: []

f8  before: []
     after: []
     right: [']']
```

Return none or return string.

CHAPTER

SIXTY

IMPORTTOOLS

60.1 Functions

60.1.1 importtools.import_structured_package

importtools.**import_structured_package**(*path*, *namespace*, *package_root_name='abjad'*)

New in version 2.9. Import public names from *path* into *namespace*.

This is the custom function that all Abjad packages use to import public classes and functions on startup.

The function will work for any package laid out like Abjad packages.

Set *package_root_name* to the root any Abjad-like package structure.

Return none.

CHAPTER

SIXTYONE

INTROSPECTIONTOOLS

61.1 Functions

61.1.1 introspectiontools.get_current_function_name

introspectiontools.**get_current_function_name**()

New in version 2.10. Get current function name:

```
>>> def foo():
...         function_name = introspectiontools.get_current_function_name()
...         print 'Function name is {!r}.'.format(function_name)
```

```
>>> foo()
Function name is 'foo'.
```

Call this function within the implementation of any ofther function.

Returns enclosing function name as a string or else none.

61.1.2 introspectiontools.klass_to_tools_package_qualified_klass_name

introspectiontools.**klass_to_tools_package_qualified_klass_name**(*klass*)

New in version 2.10. Change *klass* to tools package-qualified class name:

```
>>> introspectiontools.klass_to_tools_package_qualified_klass_name(Note)
'notetools.Note'
```

Return string.

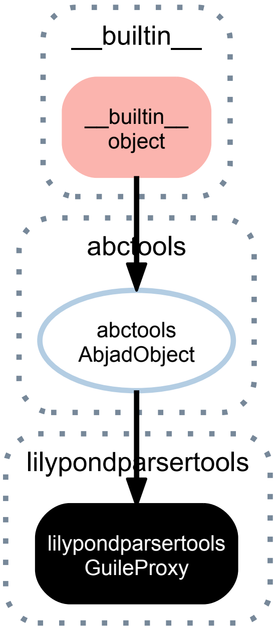
CHAPTER

SIXTYTWO

LILYPONDPARSERTOOLS

62.1 Concrete Classes

62.1.1 lilypondparsertools.GuileProxy



class `lilypondparsertools.`**`GuileProxy`**(*client*)

Emulates LilyPond music functions.

Used internally by LilyPondParser.

Not composer-safe.

Read-only properties

`GuileProxy.`**`storage_format`**

Storage format of Abjad object.

Return string.

Methods

`GuileProxy.`**`acciaccatura`**(*music*)

`GuileProxy.`**`appoggiatura`**(*music*)

GuileProxy.**bar**(*string*)

GuileProxy.**breathe**()

GuileProxy.**clef**(*string*)

GuileProxy.**grace**(*music*)

GuileProxy.**key**(*notename_pitch*, *number_list*)

GuileProxy.**language**(*string*)

GuileProxy.**makeClusters**(*music*)

GuileProxy.**mark**(*label*)

GuileProxy.**oneVoice**()

GuileProxy.**relative**(*pitch*, *music*)

GuileProxy.**skip**(*duration*)

GuileProxy.**slashedGraceContainer**(*music*)

GuileProxy.**time**(*number_list*, *fraction*)

GuileProxy.**times**(*fraction*, *music*)

GuileProxy.**transpose**(*from_pitch*, *to_pitch*, *music*)

GuileProxy.**voiceFour**()

GuileProxy.**voiceOne**()

GuileProxy.**voiceThree**()

GuileProxy.**voiceTwo**()

Special methods

GuileProxy.**__call__**(*function_name*, *args*)

GuileProxy.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

GuileProxy.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GuileProxy.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

GuileProxy.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GuileProxy.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

GuileProxy.**__ne__**(*expr*)
Defined equal to the opposite of equality.

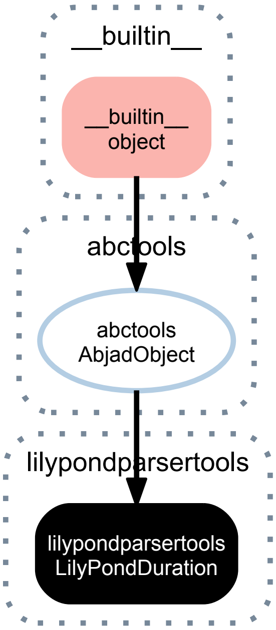
Return boolean.

GuileProxy.**__repr__**()

Interpreter representation of Abjad object.

Return string.

62.1.2 lilypondparsertools.LilyPondDuration



class lilypondparsertools.**LilyPondDuration**(*duration*, *multiplier=None*)

Model of a duration in LilyPond.

Not composer-safe.

Used internally by LilyPondParser.

Read-only properties

LilyPondDuration.**storage_format**

Storage format of Abjad object.

Return string.

Special methods

LilyPondDuration.**__eq__**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

LilyPondDuration.**__ge__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

LilyPondDuration.**__gt__**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

LilyPondDuration.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondDuration.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

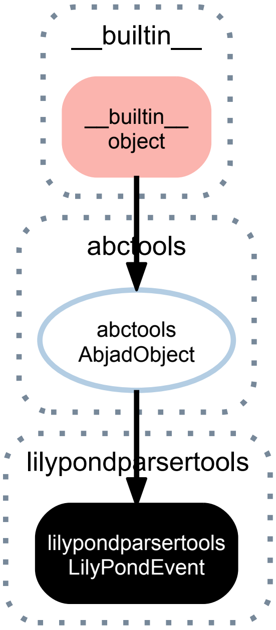
LilyPondDuration.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LilyPondDuration.**__repr__**()
Interpreter representation of Abjad object.

Return string.

62.1.3 lilypondparsertools.LilyPondEvent



class lilypondparsertools.**LilyPondEvent**(*name*, ***kwargs*)
Model of an arbitrary event in LilyPond.

Not composer-safe.

Used internally by LilyPondParser.

Read-only properties

LilyPondEvent.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

LilyPondEvent.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`LilyPondEvent.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondEvent.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`LilyPondEvent.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondEvent.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

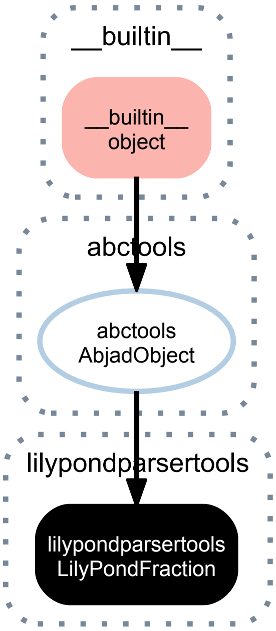
Raise exception.

`LilyPondEvent.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`LilyPondEvent.`**`__repr__`**()

62.1.4 lilypondparsertools.LilyPondFraction



class `lilypondparsertools.`**`LilyPondFraction`**(*numerator*, *denominator*)
Model of a fraction in LilyPond.

Not composer-safe.

Used internally by LilyPondParser.

Read-only properties

`LilyPondFraction.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`LilyPondFraction.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`LilyPondFraction.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondFraction.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`LilyPondFraction.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondFraction.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

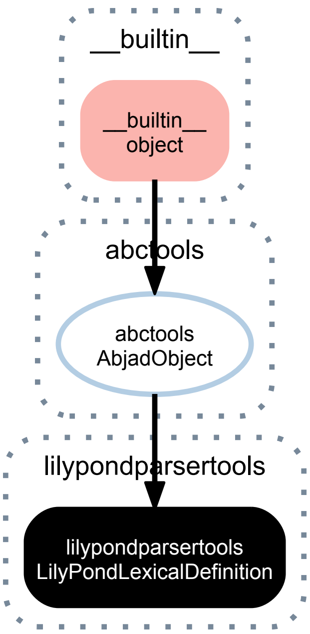
`LilyPondFraction.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`LilyPondFraction.__repr__()`
Interpreter representation of Abjad object.

Return string.

62.1.5 lilypondparsertools.LilyPondLexicalDefinition



class lilypondparsertools.**LilyPondLexicalDefinition**(*client*)
The lexical definition of LilyPond's syntax.

Effectively equivalent to LilyPond's `lexer.ll` file.

Not composer-safe.

Used internally by `LilyPondParser`.

Read-only properties

LilyPondLexicalDefinition.**storage_format**
Storage format of Abjad object.

Return string.

Methods

LilyPondLexicalDefinition.**push_signature**(*signature*, *t*)

LilyPondLexicalDefinition.**scan_bare_word**(*t*)

LilyPondLexicalDefinition.**scan_escaped_word**(*t*)

LilyPondLexicalDefinition.**t_651_a**(*t*)
(((-?[0-9]+).[0-9]*)|(-?.[0-9]+))

LilyPondLexicalDefinition.**t_651_b**(*t*)
-[0-9]+

LilyPondLexicalDefinition.**t_661**(*t*)
-.

LilyPondLexicalDefinition.**t_666**(*t*)
[0-9]+

LilyPondLexicalDefinition.**t_ANY_165**(*t*)
r

LilyPondLexicalDefinition.**t_INITIAL_643**(*t*)
[a-zA-Z200-377](((([a-zA-Z200-377]|_)|[0-9])|-)*

LilyPondLexicalDefinition.**t_INITIAL_646**(*t*)
\[a-zA-Z200-377](((([a-zA-Z200-377]|_)|[0-9])|-)*

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_210**(*t*)
%{

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_214**(*t*)
%[^{nr][^nr]*[nr]

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_216**(*t*)
%[^{nr]

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_218**(*t*)
%[nr]

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_220**(*t*)
%[^{nr][^nr]*

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_222**(*t*)
[

]

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_227**(*t*)
“

LilyPondLexicalDefinition.**t_INITIAL_markup_notes_353**(*t*)
#

LilyPondLexicalDefinition.**t_INITIAL_notes_233**(*t*)
\version[ntfr]*

LilyPondLexicalDefinition.**t_INITIAL_notes_387**(*t*)
<<

LilyPondLexicalDefinition.**t_INITIAL_notes_390**(*t*)
>>

LilyPondLexicalDefinition.**t_INITIAL_notes_396**(*t*)
<

LilyPondLexicalDefinition.**t_INITIAL_notes_399**(*t*)
>

LilyPondLexicalDefinition.**t_INITIAL_notes_686**(*t*)
\.

LilyPondLexicalDefinition.**t_error**(*t*)

LilyPondLexicalDefinition.**t_longcomment_291**(*t*)
[^%]+

LilyPondLexicalDefinition.**t_longcomment_293**(*t*)
%+[^}%]*

LilyPondLexicalDefinition.**t_longcomment_296**(*t*)
%}

LilyPondLexicalDefinition.**t_longcomment_error**(*t*)

LilyPondLexicalDefinition.**t_markup_545**(*t*)
\score

LilyPondLexicalDefinition.**t_markup_548**(*t*)
\([a-zA-Z200-377]|[-_])+

LilyPondLexicalDefinition.**t_markup_601**(*t*)
[^#{}"\ tnrf]+

LilyPondLexicalDefinition.**t_markup_error**(*t*)

LilyPondLexicalDefinition.**t_newline**(*t*)
n+

LilyPondLexicalDefinition.**t_notes_417**(*t*)
[a-zA-Z200-377]+

LilyPondLexicalDefinition.**t_notes_421**(*t*)
\[a-zA-Z200-377]+

LilyPondLexicalDefinition.**t_notes_424**(*t*)
[0-9]+/[0-9]+

LilyPondLexicalDefinition.**t_notes_428**(*t*)
[0-9]+//

LilyPondLexicalDefinition.**t_notes_428b**(*t*)
[0-9]+

LilyPondLexicalDefinition.**t_notes_433**(*t*)
\[0-9]+

LilyPondLexicalDefinition.**t_notes_error**(*t*)

LilyPondLexicalDefinition.**t_quote_440**(*t*)
[nt\"]

`LilyPondLexicalDefinition.t_quote_443(t)`
[^\"“]+

`LilyPondLexicalDefinition.t_quote_446(t)`
“

`LilyPondLexicalDefinition.t_quote_456(t)`
.

`LilyPondLexicalDefinition.t_quote_XXX(t)`
\”

`LilyPondLexicalDefinition.t_quote_error(t)`

`LilyPondLexicalDefinition.t_scheme_error(t)`

`LilyPondLexicalDefinition.t_version_242(t)`
“[^”]*”

`LilyPondLexicalDefinition.t_version_278(t)`
(.ln)

`LilyPondLexicalDefinition.t_version_341(t)`
“[^”]*

`LilyPondLexicalDefinition.t_version_error(t)`

Special methods

`LilyPondLexicalDefinition.__eq__(expr)`
True when `id(self)` equals `id(expr)`.

Return boolean.

`LilyPondLexicalDefinition.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondLexicalDefinition.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`LilyPondLexicalDefinition.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondLexicalDefinition.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

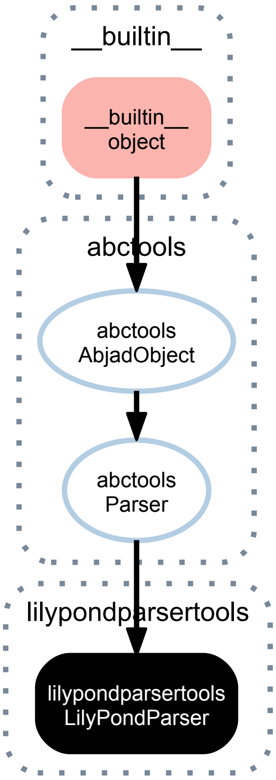
`LilyPondLexicalDefinition.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`LilyPondLexicalDefinition.__repr__()`
Interpreter representation of Abjad object.

Return string.

62.1.6 lilypondparsertools.LilyPondParser



class `lilypondparsertools.`**`LilyPondParser`**(*default_language='english'*, *debug=False*)

Parses a subset of LilyPond input syntax:

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> input = r"\new Staff { c'4 ( d'8 e' fs'2) \fermata }"
>>> result = parser(input)
>>> f(result)
\new Staff {
    c'4 (
    d'8
    e'8
    fs'2 -\fermata )
}
```

LilyPondParser defaults to English note names, but any of the other languages supported by LilyPond may be used:

```
>>> parser = lilypondparsertools.LilyPondParser('nederlands')
>>> input = '{ c des e fis }'
>>> result = parser(input)
>>> f(result)
{
    c4
    df4
    e4
    fs4
}
```

Briefly, LilyPondParser understands theses aspects of LilyPond syntax:

- Notes, chords, rests, skips and multi-measure rests
- Durations, dots, and multipliers
- All pitchnames, and octave ticks
- Simple markup (i.e. `c'4 ^ "hello!"`)

- Most articulations
- Most spanners, including beams, slurs, phrasing slurs, ties, and glissandi
- Most context types via `\new` and `\context`, as well as context ids (i.e. `\new Staff = "foo" { }`)
- Variable assignment (i.e. `global = { \time 3/4 } \new Staff { \global }`)
- **Many music functions:**
  - `\acciaccatura`
  - `\appoggiatura`
  - `\bar`
  - `\breathe`
  - `\clef`
  - `\grace`
  - `\key`
  - `\transpose`
  - `\language`
  - `\makeClusters`
  - `\mark`
  - `\oneVoice`
  - `\relative`
  - `\skip`
  - `\slashedGrace`
  - `\time`
  - `\times`
  - `\transpose`
  - `\voiceOne`, `\voiceTwo`, `\voiceThree`, `\voiceFour`

LilyPondParser currently **DOES NOT** understand many other aspects of LilyPond syntax:

- `\markup`
- `\book`, `\bookpart`, `\header`, `\layout`, `\midi` and `\paper`
- `\repeat` and `\alternative`
- Lyrics
- `\chordmode`, `\drummode` or `\figuremode`
- Property operations, such as `\override`, `\revert`, `\set`, `\unset`, and `\once`
- Music functions which generate or extensively mutate musical structures
- Embedded Scheme statements (anything beginning with `#`)

Returns LilyPondParser instance.

Read-only properties

LilyPondParser.**available_languages**

Tuple of pitch-name languages supported by LilyPondParser:

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> for language in parser.available_languages:
...     print language
catalan
deutsch
english
espanol
español
français
italiano
nederlands
norsk
portugues
suomi
svenska
vlaams
```

Return tuple.

LilyPondParser.**debug**

True if the parser runs in debugging mode.

LilyPondParser.**lexer**

The parser's PLY Lexer instance.

LilyPondParser.**lexer_rules_object**

LilyPondParser.**logger**

The parser's Logger instance.

LilyPondParser.**logger_path**

The output path for the parser's logfile.

LilyPondParser.**output_path**

The output path for files associated with the parser.

LilyPondParser.**parser**

The parser's PLY LRParser instance.

LilyPondParser.**parser_rules_object**

LilyPondParser.**pickle_path**

The output path for the parser's pickled parsing tables.

LilyPondParser.**storage_format**

Storage format of Abjad object.

Return string.

Read/write properties

LilyPondParser.**default_language**

Read/write attribute to set parser's default pitch-name language:

```
>>> parser = lilypondparsertools.LilyPondParser()
```

```
>>> parser.default_language
'english'
```

```
>>> parser('{ c df e fs }')
{c4, df4, e4, fs4}
```

```
>>> parser.default_language = 'nederlands'
>>> parser.default_language
'nederlands'
```

```
>>> parser('{ c des e fis }')
{c4, df4, e4, fs4}
```

Return string.

Methods

classmethod `LilyPondParser.`**`register_markup_function`**(*name*, *signature*)
Register a custom markup function globally with LilyPondParser:

```
>>> name = 'my-custom-markup-function'
>>> signature = ['markup?']
>>> lilypondparsertools.LilyPondParser.register_markup_function(name, signature)
```

```
>>> parser = lilypondparsertools.LilyPondParser()
>>> string = r"\markup { \my-custom-markup-function { foo bar baz } }"
>>> parser(string)
Markup((MarkupCommand('my-custom-markup-function', ['foo', 'bar', 'baz']),))
```

signature should be a sequence of zero or more type-predicate names, as understood by LilyPond. Consult LilyPond's documentation for a complete list of all understood type-predicates.

Return None

`LilyPondParser.`**`tokenize`**(*input_string*)
Tokenize *input string* and print results.

Special methods

`LilyPondParser.`**`__call__`**(*input_string*)

`LilyPondParser.`**`__eq__`**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

`LilyPondParser.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondParser.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`LilyPondParser.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondParser.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

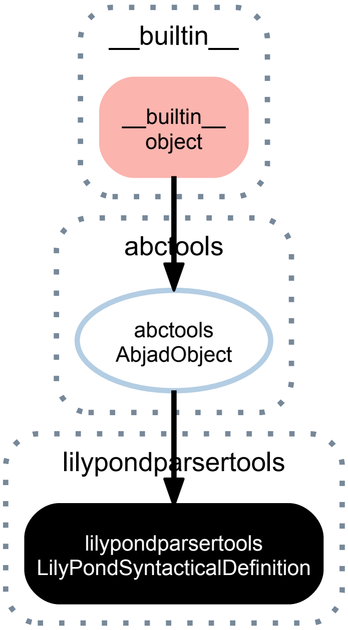
`LilyPondParser.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LilyPondParser.**__repr__**()
Interpreter representation of Abjad object.

Return string.

62.1.7 lilypondparsertools.LilyPondSyntacticalDefinition



class lilypondparsertools.**LilyPondSyntacticalDefinition**(*client*)
The syntactical definition of LilyPond's syntax.

Effectively equivalent to LilyPond's `parser.yy` file.

Not composer-safe.

Used internally by `LilyPondParser`.

Read-only properties

LilyPondSyntacticalDefinition.**storage_format**
Storage format of Abjad object.

Return string.

Methods

LilyPondSyntacticalDefinition.**p_assignment__assignment_id__Chr61__identifier_init**(*p*)
assignment : assignment_id '=' identifier_init

LilyPondSyntacticalDefinition.**p_assignment__embedded_scm**(*p*)
assignment : embedded_scm

LilyPondSyntacticalDefinition.**p_assignment_id__STRING**(*p*)
assignment_id : STRING

LilyPondSyntacticalDefinition.**p_bare_number__REAL__NUMBER_IDENTIFIER**(*p*)
bare_number : REAL NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_bare_number__UNSIGNED__NUMBER_IDENTIFIER**(*p*)
bare_number : UNSIGNED NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_bare_number__bare_number_closed**(*p*)
bare_number : bare_number_closed

LilyPondSyntacticalDefinition.**p_bare_number_closed__NUMBER_IDENTIFIER**(*p*)
bare_number_closed : NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_bare_number_closed__REAL**(*p*)
bare_number_closed : REAL

LilyPondSyntacticalDefinition.**p_bare_number_closed__UNSIGNED**(*p*)
bare_number_closed : UNSIGNED

LilyPondSyntacticalDefinition.**p_bare_unsigned__UNSIGNED**(*p*)
bare_unsigned : UNSIGNED

LilyPondSyntacticalDefinition.**p_braced_music_list__Chr123__music_list__Chr125**(*p*)
braced_music_list : '{' music_list '}'

LilyPondSyntacticalDefinition.**p_chord_body__ANGLE_OPEN__chord_body_elements__ANGLE_CLOSE**
chord_body : ANGLE_OPEN chord_body_elements ANGLE_CLOSE

LilyPondSyntacticalDefinition.**p_chord_body_element__music_function_chord_body**(*p*)
chord_body_element : music_function_chord_body

LilyPondSyntacticalDefinition.**p_chord_body_element__pitch__exclamations__questions__octa**
chord_body_element : pitch exclamations questions octave_check post_events

LilyPondSyntacticalDefinition.**p_chord_body_elements__Empty**(*p*)
chord_body_elements :

LilyPondSyntacticalDefinition.**p_chord_body_elements__chord_body_elements__chord_body_ele**
chord_body_elements : chord_body_elements chord_body_element

LilyPondSyntacticalDefinition.**p_closed_music__complex_music_prefix__closed_music**(*p*)
closed_music : complex_music_prefix closed_music

LilyPondSyntacticalDefinition.**p_closed_music__music_bare**(*p*)
closed_music : music_bare

LilyPondSyntacticalDefinition.**p_command_element__Chr124**(*p*)
command_element : '|'

LilyPondSyntacticalDefinition.**p_command_element__E_BACKSLASH**(*p*)
command_element : E_BACKSLASH

LilyPondSyntacticalDefinition.**p_command_element__command_event**(*p*)
command_element : command_event

LilyPondSyntacticalDefinition.**p_command_event__tempo_event**(*p*)
command_event : tempo_event

LilyPondSyntacticalDefinition.**p_complex_music__complex_music_prefix__music**(*p*)
complex_music : complex_music_prefix music

LilyPondSyntacticalDefinition.**p_complex_music__music_function_call**(*p*)
complex_music : music_function_call

LilyPondSyntacticalDefinition.**p_complex_music_prefix__CONTEXT__simple_string__optional_i**
complex_music_prefix : CONTEXT simple_string optional_id optional_context_mod

LilyPondSyntacticalDefinition.**p_complex_music_prefix__NEWCONTEXT__simple_string__optiona**
complex_music_prefix : NEWCONTEXT simple_string optional_id optional_context_mod

LilyPondSyntacticalDefinition.**p_composite_music__complex_music**(*p*)
composite_music : complex_music

LilyPondSyntacticalDefinition.**p_composite_music__music_bare**(*p*)
composite_music : music_bare

LilyPondSyntacticalDefinition.**p_context_change__CHANGE__STRING__Chr61__STRING**(*p*)
context_change : CHANGE STRING '=' STRING

LilyPondSyntacticalDefinition.**p_context_def_spec_block__CONTEXT__Chr123__context_def_spe**
context_def_spec_block : CONTEXT '{' context_def_spec_body '}'

LilyPondSyntacticalDefinition.**p_context_def_spec_body__CONTEXT_DEF_IDENTIFIER**(*p*)
context_def_spec_body : CONTEXT_DEF_IDENTIFIER

LilyPondSyntacticalDefinition.**p_context_def_spec_body__Empty**(*p*)
context_def_spec_body :

LilyPondSyntacticalDefinition.**p_context_def_spec_body__context_def_spec_body__context_mo**
context_def_spec_body : context_def_spec_body context_mod

LilyPondSyntacticalDefinition.**p_context_def_spec_body__context_def_spec_body__context_mo**
context_def_spec_body : context_def_spec_body context_modification

LilyPondSyntacticalDefinition.**p_context_def_spec_body__context_def_spec_body__embedded_s**
context_def_spec_body : context_def_spec_body embedded_scm

LilyPondSyntacticalDefinition.**p_context_mod__property_operation**(*p*)
context_mod : property_operation

LilyPondSyntacticalDefinition.**p_context_mod_list__Empty**(*p*)
context_mod_list :

LilyPondSyntacticalDefinition.**p_context_mod_list__context_mod_list__CONTEXT_MOD_IDENTIFI**
context_mod_list : context_mod_list CONTEXT_MOD_IDENTIFIER

LilyPondSyntacticalDefinition.**p_context_mod_list__context_mod_list__context_mod**(*p*)
context_mod_list : context_mod_list context_mod

LilyPondSyntacticalDefinition.**p_context_mod_list__context_mod_list__embedded_scm**(*p*)
context_mod_list : context_mod_list embedded_scm

LilyPondSyntacticalDefinition.**p_context_modification__CONTEXT_MOD_IDENTIFIER**(*p*)
context_modification : CONTEXT_MOD_IDENTIFIER

LilyPondSyntacticalDefinition.**p_context_modification__WITH__CONTEXT_MOD_IDENTIFIER**(*p*)
context_modification : WITH CONTEXT_MOD_IDENTIFIER

LilyPondSyntacticalDefinition.**p_context_modification__WITH__Chr123__context_mod_list__Ch**
context_modification : WITH '{' context_mod_list '}'

LilyPondSyntacticalDefinition.**p_context_modification__WITH__embedded_scm_closed**(*p*)
context_modification : WITH embedded_scm_closed

LilyPondSyntacticalDefinition.**p_context_prop_spec__simple_string**(*p*)
context_prop_spec : simple_string

LilyPondSyntacticalDefinition.**p_context_prop_spec__simple_string__Chr46__simple_string**(*p*
context_prop_spec : simple_string '.' simple_string

LilyPondSyntacticalDefinition.**p_direction_less_char__Chr126**(*p*)
direction_less_char : '~'

LilyPondSyntacticalDefinition.**p_direction_less_char__Chr40**(*p*)
direction_less_char : '('

LilyPondSyntacticalDefinition.**p_direction_less_char__Chr41**(*p*)
direction_less_char : ')'

LilyPondSyntacticalDefinition.**p_direction_less_char__Chr91**(*p*)
direction_less_char : '['

LilyPondSyntacticalDefinition.**p_direction_less_char__Chr93**(*p*)
direction_less_char : ']'

LilyPondSyntacticalDefinition.**p_direction_less_char__E_ANGLE_CLOSE**(*p*)
direction_less_char : E_ANGLE_CLOSE

LilyPondSyntacticalDefinition.**p_direction_less_char__E_ANGLE_OPEN**(*p*)
direction_less_char : E_ANGLE_OPEN

LilyPondSyntacticalDefinition.**p_direction_less_char__E_CLOSE**(*p*)
direction_less_char : E_CLOSE

LilyPondSyntacticalDefinition.**p_direction_less_char__E_EXCLAMATION**(*p*)
direction_less_char : E_EXCLAMATION

LilyPondSyntacticalDefinition.**p_direction_less_char__E_OPEN**(*p*)
direction_less_char : E_OPEN

LilyPondSyntacticalDefinition.**p_direction_less_event__EVENT_IDENTIFIER**(*p*)
direction_less_event : EVENT_IDENTIFIER

LilyPondSyntacticalDefinition.**p_direction_less_event__direction_less_char**(*p*)
direction_less_event : direction_less_char

LilyPondSyntacticalDefinition.**p_direction_less_event__event_function_event**(*p*)
direction_less_event : event_function_event

LilyPondSyntacticalDefinition.**p_direction_less_event__tremolo_type**(*p*)
direction_less_event : tremolo_type

LilyPondSyntacticalDefinition.**p_direction_reqd_event__gen_text_def**(*p*)
direction_reqd_event : gen_text_def

LilyPondSyntacticalDefinition.**p_direction_reqd_event__script_abbreviation**(*p*)
direction_reqd_event : script_abbreviation

LilyPondSyntacticalDefinition.**p_dots__Empty**(*p*)
dots :

LilyPondSyntacticalDefinition.**p_dots__dots__Chr46**(*p*)
dots : dots '.'

LilyPondSyntacticalDefinition.**p_duration_length__multiplied_duration**(*p*)
duration_length : multiplied_duration

LilyPondSyntacticalDefinition.**p_embedded_scm__embedded_scm_bare**(*p*)
embedded_scm : embedded_scm_bare

LilyPondSyntacticalDefinition.**p_embedded_scm__scm_function_call**(*p*)
embedded_scm : scm_function_call

LilyPondSyntacticalDefinition.**p_embedded_scm_arg__embedded_scm_bare_arg**(*p*)
embedded_scm_arg : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.**p_embedded_scm_arg__music_arg**(*p*)
embedded_scm_arg : music_arg

LilyPondSyntacticalDefinition.**p_embedded_scm_arg__scm_function_call**(*p*)
embedded_scm_arg : scm_function_call

LilyPondSyntacticalDefinition.**p_embedded_scm_arg_closed__closed_music**(*p*)
embedded_scm_arg_closed : closed_music

LilyPondSyntacticalDefinition.**p_embedded_scm_arg_closed__embedded_scm_bare_arg**(*p*)
embedded_scm_arg_closed : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.**p_embedded_scm_arg_closed__scm_function_call_closed**(*p*)
embedded_scm_arg_closed : scm_function_call_closed

LilyPondSyntacticalDefinition.**p_embedded_scm_bare__SCM_IDENTIFIER**(*p*)
embedded_scm_bare : SCM_IDENTIFIER

LilyPondSyntacticalDefinition.**p_embedded_scm_bare__SCM_TOKEN**(*p*)
embedded_scm_bare : SCM_TOKEN

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__STRING**(*p*)
embedded_scm_bare_arg : STRING

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__STRING_IDENTIFIER**(*p*)
embedded_scm_bare_arg : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__context_def_spec_block**(*p*)
embedded_scm_bare_arg : context_def_spec_block

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__context_modification**(*p*)
embedded_scm_bare_arg : context_modification

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__embedded_scm_bare**(*p*)
embedded_scm_bare_arg : embedded_scm_bare

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__full_markup**(*p*)
embedded_scm_bare_arg : full_markup

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__full_markup_list**(*p*)
embedded_scm_bare_arg : full_markup_list

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__output_def**(*p*)
embedded_scm_bare_arg : output_def

LilyPondSyntacticalDefinition.**p_embedded_scm_bare_arg__score_block**(*p*)
embedded_scm_bare_arg : score_block

LilyPondSyntacticalDefinition.**p_embedded_scm_chord_body__SCM_FUNCTION__music_function_ch**
embedded_scm_chord_body : SCM_FUNCTION music_function_chord_body_arglist

LilyPondSyntacticalDefinition.**p_embedded_scm_chord_body__bare_number**(*p*)
embedded_scm_chord_body : bare_number

LilyPondSyntacticalDefinition.**p_embedded_scm_chord_body__chord_body_element**(*p*)
embedded_scm_chord_body : chord_body_element

LilyPondSyntacticalDefinition.**p_embedded_scm_chord_body__embedded_scm_bare_arg**(*p*)
embedded_scm_chord_body : embedded_scm_bare_arg

LilyPondSyntacticalDefinition.**p_embedded_scm_chord_body__fraction**(*p*)
embedded_scm_chord_body : fraction

LilyPondSyntacticalDefinition.**p_embedded_scm_closed__embedded_scm_bare**(*p*)
embedded_scm_closed : embedded_scm_bare

LilyPondSyntacticalDefinition.**p_embedded_scm_closed__scm_function_call_closed**(*p*)
embedded_scm_closed : scm_function_call_closed

LilyPondSyntacticalDefinition.**p_error**(*p*)

LilyPondSyntacticalDefinition.**p_event_chord__CHORD_REPETITION__optional_notemode_duratio**
event_chord : CHORD_REPETITION optional_notemode_duration post_events

LilyPondSyntacticalDefinition.**p_event_chord__MULTI_MEASURE_REST__optional_notemode_durat**
event_chord : MULTI_MEASURE_REST optional_notemode_duration post_events

LilyPondSyntacticalDefinition.**p_event_chord__command_element**(*p*)
event_chord : command_element

LilyPondSyntacticalDefinition.**p_event_chord__note_chord_element**(*p*)
event_chord : note_chord_element

LilyPondSyntacticalDefinition.**p_event_chord__simple_chord_elements__post_events**(*p*)
event_chord : simple_chord_elements post_events

LilyPondSyntacticalDefinition.**p_event_function_event__EVENT_FUNCTION__function_arglist_c**
event_function_event : EVENT_FUNCTION function_arglist_closed

LilyPondSyntacticalDefinition.**p_exclamations__Empty**(*p*)
exclamations :

LilyPondSyntacticalDefinition.**p_exclamations__exclamations__Chr33**(*p*)
exclamations : exclamations '!'

LilyPondSyntacticalDefinition.**p_fingering__UNSIGNED**(*p*)
fingering : UNSIGNED

LilyPondSyntacticalDefinition.**p_fraction__FRACTION**(*p*)
fraction : FRACTION

LilyPondSyntacticalDefinition.**p_fraction__UNSIGNED__Chr47__UNSIGNED**(*p*)
fraction : UNSIGNED '/' UNSIGNED

LilyPondSyntacticalDefinition.**p_full_markup__MARKUP_IDENTIFIER**(*p*)
full_markup : MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.**p_full_markup__MARKUP__markup_top**(*p*)
full_markup : MARKUP markup_top

LilyPondSyntacticalDefinition.**p_full_markup_list__MARKUPLIST_IDENTIFIER**(*p*)
full_markup_list : MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.**p_full_markup_list__MARKUPLIST__markup_list**(*p*)
full_markup_list : MARKUPLIST markup_list

LilyPondSyntacticalDefinition.**p_function_arglist__function_arglist_common**(*p*)
function_arglist : function_arglist_common

LilyPondSyntacticalDefinition.**p_function_arglist__function_arglist_nonbackup**(*p*)
function_arglist : function_arglist_nonbackup

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_DURATIO**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed_keep duration_length

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_PITCH__**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_keep pitch_also_in_chords

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' REAL

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep '-' UNSIGNED

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep FRACTION

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep REAL

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep UNSIGNED

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed_keep post_event_nofinger

LilyPondSyntacticalDefinition.**p_function_arglist_backup__EXPECT_OPTIONAL__EXPECT_SCM__fu**
function_arglist_backup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_keep embedded_scm_arg_closed

LilyPondSyntacticalDefinition.**p_function_arglist_backup__function_arglist_backup__REPARS**
function_arglist_backup : function_arglist_backup REPARSE bare_number

LilyPondSyntacticalDefinition.**p_function_arglist_backup__function_arglist_backup__REPARS**
function_arglist_backup : function_arglist_backup REPARSE embedded_scm_arg_closed

LilyPondSyntacticalDefinition.**p_function_arglist_backup__function_arglist_backup__REPARS**
function_arglist_backup : function_arglist_backup REPARSE fraction

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_DURATION__function_arglist**
function_arglist_bare : EXPECT_DURATION function_arglist_closed_optional duration_length

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_NO_MORE_ARGS**(*p*)
function_arglist_bare : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_DURATION_**
function_arglist_bare : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_PITCH__fu**
function_arglist_bare : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_OPTIONAL__EXPECT_SCM__func**
function_arglist_bare : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip DEFAULT

LilyPondSyntacticalDefinition.**p_function_arglist_bare__EXPECT_PITCH__function_arglist_op**
function_arglist_bare : EXPECT_PITCH function_arglist_optional pitch_also_in_chords

LilyPondSyntacticalDefinition.**p_function_arglist_closed__function_arglist_closed_common**
function_arglist_closed : function_arglist_closed_common

LilyPondSyntacticalDefinition.**p_function_arglist_closed__function_arglist_nonbackup**(*p*)
function_arglist_closed : function_arglist_nonbackup

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional bare_number

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional fraction

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_closed_optional post_event_nofinger

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__EXPECT_SCM__function_arg**
function_arglist_closed_common : EXPECT_SCM function_arglist_optional embedded_scm_arg_closed

LilyPondSyntacticalDefinition.**p_function_arglist_closed_common__function_arglist_bare**(*p*)
function_arglist_closed_common : function_arglist_bare

LilyPondSyntacticalDefinition.**p_function_arglist_closed_keep__function_arglist_backup**(*p*)
function_arglist_closed_keep : function_arglist_backup

LilyPondSyntacticalDefinition.**p_function_arglist_closed_keep__function_arglist_closed_co**
function_arglist_closed_keep : function_arglist_closed_common

LilyPondSyntacticalDefinition.**p_function_arglist_closed_optional__EXPECT_OPTIONAL__EXPEC**
function_arglist_closed_optional : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed_optional

LilyPondSyntacticalDefinition.**p_function_arglist_closed_optional__EXPECT_OPTIONAL__EXPEC**
function_arglist_closed_optional : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_closed_optional

LilyPondSyntacticalDefinition.**p_function_arglist_closed_optional__function_arglist_backu**
function_arglist_closed_optional : function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.**p_function_arglist_closed_optional__function_arglist_close**
function_arglist_closed_optional : function_arglist_closed_keep %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_function_arglist_common__EXPECT_SCM__function_arglist_cl**
function_arglist_common : EXPECT_SCM function_arglist_closed_optional bare_number

LilyPondSyntacticalDefinition.**p_function_arglist_common__EXPECT_SCM__function_arglist_cl**
function_arglist_common : EXPECT_SCM function_arglist_closed_optional fraction

LilyPondSyntacticalDefinition.**p_function_arglist_common__EXPECT_SCM__function_arglist_cl**
function_arglist_common : EXPECT_SCM function_arglist_closed_optional post_event_nofinger

LilyPondSyntacticalDefinition.**p_function_arglist_common__EXPECT_SCM__function_arglist_op**
function_arglist_common : EXPECT_SCM function_arglist_optional embedded_scm_arg

LilyPondSyntacticalDefinition.**p_function_arglist_common__function_arglist_bare**(*p*)
function_arglist_common : function_arglist_bare

LilyPondSyntacticalDefinition.**p_function_arglist_common__function_arglist_common_minus**(*p*
function_arglist_common : function_arglist_common_minus

LilyPondSyntacticalDefinition.**p_function_arglist_common_minus__EXPECT_SCM__function_argl**
function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_function_arglist_common_minus__EXPECT_SCM__function_argl**
function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' REAL

LilyPondSyntacticalDefinition.**p_function_arglist_common_minus__EXPECT_SCM__function_argl**
function_arglist_common_minus : EXPECT_SCM function_arglist_closed_optional '-' UNSIGNED

LilyPondSyntacticalDefinition.**p_function_arglist_common_minus__function_arglist_common_m**
function_arglist_common_minus : function_arglist_common_minus REPARSE bare_number

LilyPondSyntacticalDefinition.**p_function_arglist_keep__function_arglist_backup**(*p*)
function_arglist_keep : function_arglist_backup

LilyPondSyntacticalDefinition.**p_function_arglist_keep__function_arglist_common**(*p*)
function_arglist_keep : function_arglist_common

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_DURA**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_closed duration_length

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_PITC**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_PITCH function_arglist pitch_also_in_chords

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist embedded_scm_arg_closed

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' NUMBER_IDENTIFIER

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' REAL

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed '-' UNSIGNED

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed FRACTION

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed bare_number_closed

LilyPondSyntacticalDefinition.**p_function_arglist_nonbackup__EXPECT_OPTIONAL__EXPECT_SCM_**
function_arglist_nonbackup : EXPECT_OPTIONAL EXPECT_SCM function_arglist_closed post_event_nofinger

LilyPondSyntacticalDefinition.**p_function_arglist_optional__EXPECT_OPTIONAL__EXPECT_DURAT**
function_arglist_optional : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_optional

LilyPondSyntacticalDefinition.**p_function_arglist_optional__EXPECT_OPTIONAL__EXPECT_PITCH**
function_arglist_optional : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_optional

LilyPondSyntacticalDefinition.**p_function_arglist_optional__function_arglist_backup__BACK**
function_arglist_optional : function_arglist_backup BACKUP

LilyPondSyntacticalDefinition.**p_function_arglist_optional__function_arglist_keep**(*p*)
function_arglist_optional : function_arglist_keep %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_DURATION_**
function_arglist_skip : EXPECT_OPTIONAL EXPECT_DURATION function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_PITCH__fu**
function_arglist_skip : EXPECT_OPTIONAL EXPECT_PITCH function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_function_arglist_skip__EXPECT_OPTIONAL__EXPECT_SCM__func**
function_arglist_skip : EXPECT_OPTIONAL EXPECT_SCM function_arglist_skip %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_function_arglist_skip__function_arglist_common**(*p*)
function_arglist_skip : function_arglist_common

LilyPondSyntacticalDefinition.**p_gen_text_def__full_markup**(*p*)
gen_text_def : full_markup

LilyPondSyntacticalDefinition.**p_gen_text_def__simple_string**(*p*)
gen_text_def : simple_string

LilyPondSyntacticalDefinition.**p_grouped_music_list__sequential_music**(*p*)
grouped_music_list : sequential_music

LilyPondSyntacticalDefinition.**p_grouped_music_list__simultaneous_music**(*p*)
grouped_music_list : simultaneous_music

LilyPondSyntacticalDefinition.**p_identifier_init__context_def_spec_block**(*p*)
identifier_init : context_def_spec_block

LilyPondSyntacticalDefinition.**p_identifier_init__context_modification**(*p*)
identifier_init : context_modification

LilyPondSyntacticalDefinition.**p_identifier_init__embedded_scm**(*p*)
identifier_init : embedded_scm

LilyPondSyntacticalDefinition.**p_identifier_init__full_markup**(*p*)
identifier_init : full_markup

LilyPondSyntacticalDefinition.**p_identifier_init__full_markup_list**(*p*)
identifier_init : full_markup_list

LilyPondSyntacticalDefinition.**p_identifier_init__music**(*p*)
identifier_init : music

LilyPondSyntacticalDefinition.**p_identifier_init__number_expression**(*p*)
identifier_init : number_expression

LilyPondSyntacticalDefinition.**p_identifier_init__output_def**(*p*)
identifier_init : output_def

LilyPondSyntacticalDefinition.**p_identifier_init__post_event_nofinger**(*p*)
identifier_init : post_event_nofinger

LilyPondSyntacticalDefinition.**p_identifier_init__score_block**(*p*)
identifier_init : score_block

LilyPondSyntacticalDefinition.**p_identifier_init__string**(*p*)
identifier_init : string

LilyPondSyntacticalDefinition.**p_lilypond__Empty**(*p*)
lilypond :

LilyPondSyntacticalDefinition.**p_lilypond__lilypond__assignment**(*p*)
lilypond : lilypond assignment

LilyPondSyntacticalDefinition.**p_lilypond__lilypond__error**(*p*)
lilypond : lilypond error

LilyPondSyntacticalDefinition.**p_lilypond__lilypond__toplevel_expression**(*p*)
lilypond : lilypond toplevel_expression

LilyPondSyntacticalDefinition.**p_lilypond_header__HEADER__Chr123__lilypond_header_body__C**
lilypond_header : HEADER '{' lilypond_header_body '}'

LilyPondSyntacticalDefinition.**p_lilypond_header_body__Empty**(*p*)
lilypond_header_body :

LilyPondSyntacticalDefinition.**p_lilypond_header_body__lilypond_header_body__assignment**(*p*
lilypond_header_body : lilypond_header_body assignment

LilyPondSyntacticalDefinition.**p_markup__markup_head_1_list__simple_markup**(*p*)
markup : markup_head_1_list simple_markup

LilyPondSyntacticalDefinition.**p_markup__simple_markup**(*p*)
markup : simple_markup

LilyPondSyntacticalDefinition.**p_markup_braced_list__Chr123__markup_braced_list_body__Chr**
markup_braced_list : '{' markup_braced_list_body '}'

LilyPondSyntacticalDefinition.**p_markup_braced_list_body__Empty**(*p*)
markup_braced_list_body :

LilyPondSyntacticalDefinition.**p_markup_braced_list_body__markup_braced_list_body__markup**
markup_braced_list_body : markup_braced_list_body markup

LilyPondSyntacticalDefinition.**p_markup_braced_list_body__markup_braced_list_body__markup**
markup_braced_list_body : markup_braced_list_body markup_list

LilyPondSyntacticalDefinition.**p_markup_command_basic_arguments__EXPECT_MARKUP_LIST__mark**
markup_command_basic_arguments : EXPECT_MARKUP_LIST markup_command_list_arguments markup_list

LilyPondSyntacticalDefinition.**p_markup_command_basic_arguments__EXPECT_NO_MORE_ARGS**(*p*)
markup_command_basic_arguments : EXPECT_NO_MORE_ARGS

LilyPondSyntacticalDefinition.**p_markup_command_basic_arguments__EXPECT_SCM__markup_comma**
markup_command_basic_arguments : EXPECT_SCM markup_command_list_arguments embedded_scm_closed

LilyPondSyntacticalDefinition.**p_markup_command_list__MARKUP_LIST_FUNCTION__markup_comman**
markup_command_list : MARKUP_LIST_FUNCTION markup_command_list_arguments

LilyPondSyntacticalDefinition.**p_markup_command_list_arguments__EXPECT_MARKUP__markup_com**
markup_command_list_arguments : EXPECT_MARKUP markup_command_list_arguments markup

LilyPondSyntacticalDefinition.**p_markup_command_list_arguments__markup_command_basic_argu**
markup_command_list_arguments : markup_command_basic_arguments

LilyPondSyntacticalDefinition.**p_markup_composed_list__markup_head_1_list__markup_braced_**
markup_composed_list : markup_head_1_list markup_braced_list

LilyPondSyntacticalDefinition.**p_markup_head_1_item__MARKUP_FUNCTION__EXPECT_MARKUP__mark**
markup_head_1_item : MARKUP_FUNCTION EXPECT_MARKUP markup_command_list_arguments

LilyPondSyntacticalDefinition.**p_markup_head_1_list__markup_head_1_item**(*p*)
markup_head_1_list : markup_head_1_item

LilyPondSyntacticalDefinition.**p_markup_head_1_list__markup_head_1_list__markup_head_1_it**
markup_head_1_list : markup_head_1_list markup_head_1_item

LilyPondSyntacticalDefinition.**p_markup_list__MARKUPLIST_IDENTIFIER**(*p*)
markup_list : MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.**p_markup_list__markup_braced_list**(*p*)
markup_list : markup_braced_list

LilyPondSyntacticalDefinition.**p_markup_list__markup_command_list**(*p*)
markup_list : markup_command_list

LilyPondSyntacticalDefinition.**p_markup_list__markup_composed_list**(*p*)
markup_list : markup_composed_list

LilyPondSyntacticalDefinition.**p_markup_list__markup_scm__MARKUPLIST_IDENTIFIER**(*p*)
markup_list : markup_scm MARKUPLIST_IDENTIFIER

LilyPondSyntacticalDefinition.**p_markup_scm__embedded_scm_bare__BACKUP**(*p*)
markup_scm : embedded_scm_bare BACKUP

LilyPondSyntacticalDefinition.**p_markup_top__markup_head_1_list__simple_markup**(*p*)
markup_top : markup_head_1_list simple_markup

LilyPondSyntacticalDefinition.**p_markup_top__markup_list**(*p*)
markup_top : markup_list

LilyPondSyntacticalDefinition.**p_markup_top__simple_markup**(*p*)
markup_top : simple_markup

LilyPondSyntacticalDefinition.**p_multiplied_duration__multiplied_duration__Chr42__FRACTIO**
multiplied_duration : multiplied_duration '*' FRACTION

LilyPondSyntacticalDefinition.**p_multiplied_duration__multiplied_duration__Chr42__bare_un**
multiplied_duration : multiplied_duration '*' bare_unsigned

LilyPondSyntacticalDefinition.**p_multiplied_duration__steno_duration**(*p*)
multiplied_duration : steno_duration

LilyPondSyntacticalDefinition.**p_music__composite_music**(*p*)
music : composite_music %prec COMPOSITE

LilyPondSyntacticalDefinition.**p_music__simple_music**(*p*)
music : simple_music

LilyPondSyntacticalDefinition.**p_music_arg__composite_music**(*p*)
music_arg : composite_music %prec COMPOSITE

LilyPondSyntacticalDefinition.**p_music_arg__simple_music**(*p*)
music_arg : simple_music

LilyPondSyntacticalDefinition.**p_music_bare__MUSIC_IDENTIFIER**(*p*)
music_bare : MUSIC_IDENTIFIER

LilyPondSyntacticalDefinition.**p_music_bare__grouped_music_list**(*p*)
music_bare : grouped_music_list

LilyPondSyntacticalDefinition.**p_music_function_call__MUSIC_FUNCTION__function_arglist**(*p*)
music_function_call : MUSIC_FUNCTION function_arglist

LilyPondSyntacticalDefinition.**p_music_function_chord_body__MUSIC_FUNCTION__music_functio**
music_function_chord_body : MUSIC_FUNCTION music_function_chord_body_arglist

LilyPondSyntacticalDefinition.**p_music_function_chord_body_arglist__EXPECT_SCM__music_fun**
music_function_chord_body_arglist : EXPECT_SCM music_function_chord_body_arglist embedded_scm_chord_body

LilyPondSyntacticalDefinition.**p_music_function_chord_body_arglist__function_arglist_bare**
music_function_chord_body_arglist : function_arglist_bare

LilyPondSyntacticalDefinition.**p_music_function_event__MUSIC_FUNCTION__function_arglist_c**
music_function_event : MUSIC_FUNCTION function_arglist_closed

LilyPondSyntacticalDefinition.**p_music_list__Empty**(*p*)
music_list :

LilyPondSyntacticalDefinition.**p_music_list__music_list__embedded_scm**(*p*)
music_list : music_list embedded_scm

LilyPondSyntacticalDefinition.**p_music_list__music_list__error**(*p*)
music_list : music_list error

LilyPondSyntacticalDefinition.**p_music_list__music_list__music**(*p*)
music_list : music_list music

LilyPondSyntacticalDefinition.**p_music_property_def__simple_music_property_def**(*p*)
music_property_def : simple_music_property_def

LilyPondSyntacticalDefinition.**p_note_chord_element__chord_body__optional_notemode_durati**
note_chord_element : chord_body optional_notemode_duration post_events

LilyPondSyntacticalDefinition.**p_number_expression__number_expression__Chr43__number_term**
number_expression : number_expression '+' number_term

LilyPondSyntacticalDefinition.**p_number_expression__number_expression__Chr45__number_term**
number_expression : number_expression '-' number_term

LilyPondSyntacticalDefinition.**p_number_expression__number_term**(*p*)
number_expression : number_term

LilyPondSyntacticalDefinition.**p_number_factor__Chr45__number_factor**(*p*)
number_factor : '-' number_factor

LilyPondSyntacticalDefinition.**p_number_factor__bare_number**(*p*)
number_factor : bare_number

LilyPondSyntacticalDefinition.**p_number_term__number_factor**(*p*)
number_term : number_factor

LilyPondSyntacticalDefinition.**p_number_term__number_factor__Chr42__number_factor**(*p*)
number_term : number_factor '*' number_factor

LilyPondSyntacticalDefinition.**p_number_term__number_factor__Chr47__number_factor**(*p*)
number_term : number_factor '/' number_factor

LilyPondSyntacticalDefinition.**p_octave_check__Chr61**(*p*)
octave_check : '='

LilyPondSyntacticalDefinition.**p_octave_check__Chr61__sub_quotes**(*p*)
octave_check : '=' sub_quotes

LilyPondSyntacticalDefinition.**p_octave_check__Chr61__sup_quotes**(*p*)
octave_check : '=' sup_quotes

LilyPondSyntacticalDefinition.**p_octave_check__Empty**(*p*)
octave_check :

LilyPondSyntacticalDefinition.**p_optional_context_mod__Empty**(*p*)
optional_context_mod :

LilyPondSyntacticalDefinition.**p_optional_context_mod__context_modification**(*p*)
optional_context_mod : context_modification

LilyPondSyntacticalDefinition.**p_optional_id__Chr61__simple_string**(*p*)
optional_id : '=' simple_string

LilyPondSyntacticalDefinition.**p_optional_id__Empty**(*p*)
optional_id :

LilyPondSyntacticalDefinition.**p_optional_notemode_duration__Empty**(*p*)
optional_notemode_duration :

LilyPondSyntacticalDefinition.**p_optional_notemode_duration__multiplied_duration**(*p*)
optional_notemode_duration : multiplied_duration

LilyPondSyntacticalDefinition.**p_optional_rest__Empty**(*p*)
optional_rest :

LilyPondSyntacticalDefinition.**p_optional_rest__REST**(*p*)
optional_rest : REST

LilyPondSyntacticalDefinition.**p_output_def__output_def_body__Chr125**(*p*)
output_def : output_def_body '}'

LilyPondSyntacticalDefinition.**p_output_def_body__output_def_body__assignment**(*p*)
output_def_body : output_def_body assignment

LilyPondSyntacticalDefinition.**p_output_def_body__output_def_head_with_mode_switch__Chr12**
output_def_body : output_def_head_with_mode_switch '{'

LilyPondSyntacticalDefinition.**p_output_def_body__output_def_head_with_mode_switch__Chr12**
output_def_body : output_def_head_with_mode_switch '{' OUTPUT_DEF_IDENTIFIER

LilyPondSyntacticalDefinition.**p_output_def_head__LAYOUT**(*p*)
output_def_head : LAYOUT

LilyPondSyntacticalDefinition.**p_output_def_head__MIDI**(*p*)
output_def_head : MIDI

LilyPondSyntacticalDefinition.**p_output_def_head__PAPER**(*p*)
output_def_head : PAPER

LilyPondSyntacticalDefinition.**p_output_def_head_with_mode_switch__output_def_head**(*p*)
output_def_head_with_mode_switch : output_def_head

LilyPondSyntacticalDefinition.**p_pitch__PITCH_IDENTIFIER**(*p*)
pitch : PITCH_IDENTIFIER

LilyPondSyntacticalDefinition.**p_pitch__steno_pitch**(*p*)
pitch : steno_pitch

LilyPondSyntacticalDefinition.**p_pitch_also_in_chords__pitch**(*p*)
pitch_also_in_chords : pitch

LilyPondSyntacticalDefinition.**p_pitch_also_in_chords__steno_tonic_pitch**(*p*)
pitch_also_in_chords : steno_tonic_pitch

LilyPondSyntacticalDefinition.**p_post_event__Chr45__fingering**(*p*)
post_event : '-' fingering

LilyPondSyntacticalDefinition.**p_post_event__post_event_nofinger**(*p*)
post_event : post_event_nofinger

LilyPondSyntacticalDefinition.**p_post_event_nofinger__Chr94__fingering**(*p*)
post_event_nofinger : '^' fingering

LilyPondSyntacticalDefinition.**p_post_event_nofinger__Chr95__fingering**(*p*)
post_event_nofinger : '_' fingering

LilyPondSyntacticalDefinition.**p_post_event_nofinger__EXTENDER**(*p*)
post_event_nofinger : EXTENDER

LilyPondSyntacticalDefinition.**p_post_event_nofinger__HYPHEN**(*p*)
post_event_nofinger : HYPHEN

LilyPondSyntacticalDefinition.**p_post_event_nofinger__direction_less_event**(*p*)
post_event_nofinger : direction_less_event

LilyPondSyntacticalDefinition.**p_post_event_nofinger__script_dir__direction_less_event**(*p*)
post_event_nofinger : script_dir direction_less_event

LilyPondSyntacticalDefinition.**p_post_event_nofinger__script_dir__direction_reqd_event**(*p*)
post_event_nofinger : script_dir direction_reqd_event

LilyPondSyntacticalDefinition.**p_post_event_nofinger__script_dir__music_function_event**(*p*)
post_event_nofinger : script_dir music_function_event

LilyPondSyntacticalDefinition.**p_post_event_nofinger__string_number_event**(*p*)
post_event_nofinger : string_number_event

LilyPondSyntacticalDefinition.**p_post_events__Empty**(*p*)
post_events :

LilyPondSyntacticalDefinition.**p_post_events__post_events__post_event**(*p*)
post_events : post_events post_event

LilyPondSyntacticalDefinition.**p_property_operation__OVERRIDE__simple_string__property_pa**
property_operation : OVERRIDE simple_string property_path '=' scalar

LilyPondSyntacticalDefinition.**p_property_operation__REVERT__simple_string__embedded_scm**
property_operation : REVERT simple_string embedded_scm

LilyPondSyntacticalDefinition.**p_property_operation__STRING__Chr61__scalar**(*p*)
property_operation : STRING '=' scalar

LilyPondSyntacticalDefinition.**p_property_operation__UNSET__simple_string**(*p*)
property_operation : UNSET simple_string

LilyPondSyntacticalDefinition.**p_property_path__property_path_revved**(*p*)
property_path : property_path_revved

LilyPondSyntacticalDefinition.**p_property_path_revved__embedded_scm_closed**(*p*)
property_path_revved : embedded_scm_closed

LilyPondSyntacticalDefinition.**p_property_path_revved__property_path_revved__embedded_scm**
property_path_revved : property_path_revved embedded_scm_closed

LilyPondSyntacticalDefinition.**p_questions__Empty**(*p*)
questions :

LilyPondSyntacticalDefinition.**p_questions__questions__Chr63**(*p*)
questions : questions '?'

LilyPondSyntacticalDefinition.**p_scalar__bare_number**(*p*)
scalar : bare_number

LilyPondSyntacticalDefinition.**p_scalar__embedded_scm_arg**(*p*)
scalar : embedded_scm_arg

LilyPondSyntacticalDefinition.**p_scalar_closed__bare_number**(*p*)
scalar_closed : bare_number

LilyPondSyntacticalDefinition.**p_scalar_closed__embedded_scm_arg_closed**(*p*)
scalar_closed : embedded_scm_arg_closed

LilyPondSyntacticalDefinition.**p_scm_function_call__SCM_FUNCTION__function_arglist**(*p*)
scm_function_call : SCM_FUNCTION function_arglist

LilyPondSyntacticalDefinition.**p_scm_function_call_closed__SCM_FUNCTION__function_arglist**
scm_function_call_closed : SCM_FUNCTION function_arglist_closed %prec FUNCTION_ARGLIST

LilyPondSyntacticalDefinition.**p_score_block__SCORE__Chr123__score_body__Chr125**(*p*)
score_block : SCORE '{' score_body '}'

LilyPondSyntacticalDefinition.**p_score_body__SCORE_IDENTIFIER**(*p*)
score_body : SCORE_IDENTIFIER

LilyPondSyntacticalDefinition.**p_score_body__music**(*p*)
score_body : music

LilyPondSyntacticalDefinition.**p_score_body__score_body__lilypond_header**(*p*)
score_body : score_body lilypond_header

LilyPondSyntacticalDefinition.**p_score_body__score_body__output_def**(*p*)
score_body : score_body output_def

LilyPondSyntacticalDefinition.**p_script_abbreviation__ANGLE_CLOSE**(*p*)
script_abbreviation : ANGLE_CLOSE

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr124**(*p*)
script_abbreviation : '|'

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr43**(*p*)
script_abbreviation : '+'

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr45**(*p*)
script_abbreviation : '-'

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr46**(*p*)
script_abbreviation : '.'

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr94**(*p*)
script_abbreviation : '^'

LilyPondSyntacticalDefinition.**p_script_abbreviation__Chr95**(*p*)
script_abbreviation : '_'

LilyPondSyntacticalDefinition.**p_script_dir__Chr45**(*p*)
script_dir : '-'

LilyPondSyntacticalDefinition.**p_script_dir__Chr94**(*p*)
script_dir : '^'

LilyPondSyntacticalDefinition.**p_script_dir__Chr95**(*p*)
script_dir : '_'

LilyPondSyntacticalDefinition.**p_sequential_music__SEQUENTIAL__braced_music_list**(*p*)
sequential_music : SEQUENTIAL braced_music_list

LilyPondSyntacticalDefinition.**p_sequential_music__braced_music_list**(*p*)
sequential_music : braced_music_list

LilyPondSyntacticalDefinition.**p_simple_chord_elements__simple_element** (*p*)
simple_chord_elements : simple_element

LilyPondSyntacticalDefinition.**p_simple_element__RESTNAME__optional_notemode_duration** (*p*)
simple_element : RESTNAME optional_notemode_duration

LilyPondSyntacticalDefinition.**p_simple_element__pitch__exclamations__questions__octave_c**
simple_element : pitch exclamations questions octave_check optional_notemode_duration optional_rest

LilyPondSyntacticalDefinition.**p_simple_markup__MARKUP_FUNCTION__markup_command_basic_arg**
simple_markup : MARKUP_FUNCTION markup_command_basic_arguments

LilyPondSyntacticalDefinition.**p_simple_markup__MARKUP_IDENTIFIER** (*p*)
simple_markup : MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.**p_simple_markup__SCORE__Chr123__score_body__Chr125** (*p*)
simple_markup : SCORE '{' score_body '}'

LilyPondSyntacticalDefinition.**p_simple_markup__STRING** (*p*)
simple_markup : STRING

LilyPondSyntacticalDefinition.**p_simple_markup__STRING_IDENTIFIER** (*p*)
simple_markup : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.**p_simple_markup__markup_scm__MARKUP_IDENTIFIER** (*p*)
simple_markup : markup_scm MARKUP_IDENTIFIER

LilyPondSyntacticalDefinition.**p_simple_music__context_change** (*p*)
simple_music : context_change

LilyPondSyntacticalDefinition.**p_simple_music__event_chord** (*p*)
simple_music : event_chord

LilyPondSyntacticalDefinition.**p_simple_music__music_property_def** (*p*)
simple_music : music_property_def

LilyPondSyntacticalDefinition.**p_simple_music_property_def__OVERRIDE__context_prop_spec__**
simple_music_property_def : OVERRIDE context_prop_spec property_path '=' scalar

LilyPondSyntacticalDefinition.**p_simple_music_property_def__REVERT__context_prop_spec__em**
simple_music_property_def : REVERT context_prop_spec embedded_scm

LilyPondSyntacticalDefinition.**p_simple_music_property_def__SET__context_prop_spec__Chr61**
simple_music_property_def : SET context_prop_spec '=' scalar

LilyPondSyntacticalDefinition.**p_simple_music_property_def__UNSET__context_prop_spec** (*p*)
simple_music_property_def : UNSET context_prop_spec

LilyPondSyntacticalDefinition.**p_simple_string__STRING** (*p*)
simple_string : STRING

LilyPondSyntacticalDefinition.**p_simple_string__STRING_IDENTIFIER** (*p*)
simple_string : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.**p_simultaneous_music__DOUBLE_ANGLE_OPEN__music_list__DOUBL**
simultaneous_music : DOUBLE_ANGLE_OPEN music_list DOUBLE_ANGLE_CLOSE

LilyPondSyntacticalDefinition.**p_simultaneous_music__SIMULTANEOUS__braced_music_list** (*p*)
simultaneous_music : SIMULTANEOUS braced_music_list

LilyPondSyntacticalDefinition.**p_start_symbol__lilypond** (*p*)
start_symbol : lilypond

LilyPondSyntacticalDefinition.**p_steno_duration__DURATION_IDENTIFIER__dots** (*p*)
steno_duration : DURATION_IDENTIFIER dots

LilyPondSyntacticalDefinition.**p_steno_duration__bare_unsigned__dots** (*p*)
steno_duration : bare_unsigned dots

LilyPondSyntacticalDefinition.**p_steno_pitch__NOTENAME_PITCH**(*p*)
steno_pitch : NOTENAME_PITCH

LilyPondSyntacticalDefinition.**p_steno_pitch__NOTENAME_PITCH__sub_quotes**(*p*)
steno_pitch : NOTENAME_PITCH sub_quotes

LilyPondSyntacticalDefinition.**p_steno_pitch__NOTENAME_PITCH__sup_quotes**(*p*)
steno_pitch : NOTENAME_PITCH sup_quotes

LilyPondSyntacticalDefinition.**p_steno_tonic_pitch__TONICNAME_PITCH**(*p*)
steno_tonic_pitch : TONICNAME_PITCH

LilyPondSyntacticalDefinition.**p_steno_tonic_pitch__TONICNAME_PITCH__sub_quotes**(*p*)
steno_tonic_pitch : TONICNAME_PITCH sub_quotes

LilyPondSyntacticalDefinition.**p_steno_tonic_pitch__TONICNAME_PITCH__sup_quotes**(*p*)
steno_tonic_pitch : TONICNAME_PITCH sup_quotes

LilyPondSyntacticalDefinition.**p_string__STRING**(*p*)
string : STRING

LilyPondSyntacticalDefinition.**p_string__STRING_IDENTIFIER**(*p*)
string : STRING_IDENTIFIER

LilyPondSyntacticalDefinition.**p_string__string__Chr43__string**(*p*)
string : string '+' string

LilyPondSyntacticalDefinition.**p_string_number_event__E_UNSIGNED**(*p*)
string_number_event : E_UNSIGNED

LilyPondSyntacticalDefinition.**p_sub_quotes__Chr44**(*p*)
sub_quotes : ','

LilyPondSyntacticalDefinition.**p_sub_quotes__sub_quotes__Chr44**(*p*)
sub_quotes : sub_quotes ','

LilyPondSyntacticalDefinition.**p_sup_quotes__Chr39**(*p*)
sup_quotes : ''‘

LilyPondSyntacticalDefinition.**p_sup_quotes__sup_quotes__Chr39**(*p*)
sup_quotes : sup_quotes ''‘

LilyPondSyntacticalDefinition.**p_tempo_event__TEMPO__scalar**(*p*)
tempo_event : TEMPO scalar

LilyPondSyntacticalDefinition.**p_tempo_event__TEMPO__scalar_closed__steno_duration__Chr61**
tempo_event : TEMPO scalar_closed steno_duration '=' tempo_range

LilyPondSyntacticalDefinition.**p_tempo_event__TEMPO__steno_duration__Chr61__tempo_range**(*p*
tempo_event : TEMPO steno_duration '=' tempo_range

LilyPondSyntacticalDefinition.**p_tempo_range__bare_unsigned**(*p*)
tempo_range : bare_unsigned

LilyPondSyntacticalDefinition.**p_tempo_range__bare_unsigned__Chr126__bare_unsigned**(*p*)
tempo_range : bare_unsigned '~' bare_unsigned

LilyPondSyntacticalDefinition.**p_toplevel_expression__composite_music**(*p*)
toplevel_expression : composite_music

LilyPondSyntacticalDefinition.**p_toplevel_expression__full_markup**(*p*)
toplevel_expression : full_markup

LilyPondSyntacticalDefinition.**p_toplevel_expression__full_markup_list**(*p*)
toplevel_expression : full_markup_list

LilyPondSyntacticalDefinition.**p_toplevel_expression__lilypond_header**(*p*)
toplevel_expression : lilypond_header

LilyPondSyntacticalDefinition.**p_toplevel_expression__output_def**(*p*)
toplevel_expression : output_def

LilyPondSyntacticalDefinition.**p_toplevel_expression__score_block**(*p*)
toplevel_expression : score_block

LilyPondSyntacticalDefinition.**p_tremolo_type__Chr58**(*p*)
tremolo_type : ':'

LilyPondSyntacticalDefinition.**p_tremolo_type__Chr58__bare_unsigned**(*p*)
tremolo_type : ':' bare_unsigned

Special methods

LilyPondSyntacticalDefinition.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

LilyPondSyntacticalDefinition.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondSyntacticalDefinition.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondSyntacticalDefinition.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondSyntacticalDefinition.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

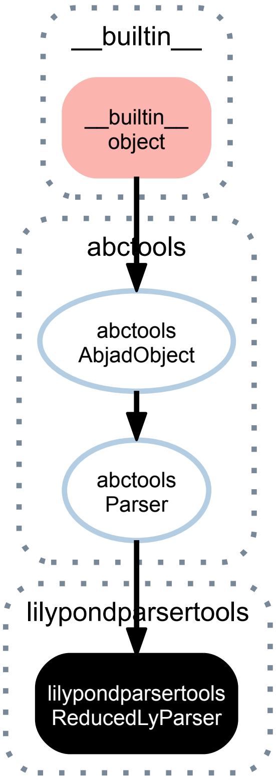
LilyPondSyntacticalDefinition.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

LilyPondSyntacticalDefinition.**__repr__**()
Interpreter representation of Abjad object.

Return string.

62.1.8 lilypondparsertools.ReducedLyParser



class lilypondparsertools.**ReducedLyParser**(*debug=False*)

Parses the "reduced-ly" syntax, a modified subset of LilyPond syntax:

```
>>> parser = lilypondparsertools.ReducedLyParser()
```

Understands LilyPond-like representation of notes, chords and rests:

```
>>> string = "c'4 r8. <b d' fs'>16"
>>> result = parser(string)
>>> f(result)
{
    c'4
    r8.
    <b d' fs'>16
}
```

Also parses bare duration as notes on middle-C, and negative bare durations as rests:

```
>>> string = '4 -8 16. -32'
>>> result = parser(string)
>>> f(result)
{
    c'4
    r8
    c'16.
    r32
}
```

Note that the leaf syntax is greedy, and therefore duration specifiers following pitch specifiers will be treated as part of the same expression. The following produces 2 leaves, rather than 3:

```
>>> string = "4 d' 4"
>>> result = parser(string)
>>> f(result)
{
    c'4
    d'4
}
```

Understands LilyPond-like default durations:

```
>>> string = "c'4 d' e' f'"
>>> result = parser(string)
>>> f(result)
{
    c'4
    d'4
    e'4
    f'4
}
```

Also understands various types of container specifications.

Can create arbitrarily nested tuplets:

```
>>> string = "2/3 { 4 4 3/5 { 8 8 8 } }"
>>> result = parser(string)
>>> f(result)
\times 2/3 {
    c'4
    c'4
    \fraction \times 3/5 {
        c'8
        c'8
        c'8
    }
}
```

Can also create empty *FixedDurationContainers*:

```
>>> string = '{1/4} {3/4}'
>>> result = parser(string)
>>> for x in result: x
...
FixedDurationContainer(Duration(1, 4), [])
FixedDurationContainer(Duration(3, 4), [])
```

Can create measures too:

```
>>> string = '| 4/4 4 4 4 4 || 3/8 8 8 8 |'
>>> result = parser(string)
>>> for x in result: x
...
Measure(4/4, [c'4, c'4, c'4, c'4])
Measure(3/8, [c'8, c'8, c'8])
```

Finally, understands ties, slurs and beams:

```
>>> string = 'c16 [ ( d ~ d ) f ]'
>>> result = parser(string)
>>> f(result)
{
    c16 [ (
    d16 ~
    d16 )
    f16 ]
}
```

Return *ReducedLyParser* instance.

Read-only properties

`ReducedLyParser.`**`debug`**
True if the parser runs in debugging mode.

`ReducedLyParser.`**`lexer`**
The parser's PLY Lexer instance.

`ReducedLyParser.`**`lexer_rules_object`**

ReducedLyParser.**logger**
The parser's Logger instance.

ReducedLyParser.**logger_path**
The output path for the parser's logfile.

ReducedLyParser.**output_path**
The output path for files associated with the parser.

ReducedLyParser.**parser**
The parser's PLY LRParser instance.

ReducedLyParser.**parser_rules_object**

ReducedLyParser.**pickle_path**
The output path for the parser's pickled parsing tables.

ReducedLyParser.**storage_format**
Storage format of Abjad object.

Return string.

Methods

ReducedLyParser.**p_apostrophes__APOSTROPHE**(*p*)
apostrophes : APOSTROPHE

ReducedLyParser.**p_apostrophes__apostrophes__APOSTROPHE**(*p*)
apostrophes : apostrophes APOSTROPHE

ReducedLyParser.**p_beam__BRACKET_L**(*p*)
beam : BRACKET_L

ReducedLyParser.**p_beam__BRACKET_R**(*p*)
beam : BRACKET_R

ReducedLyParser.**p_chord_body__chord_pitches**(*p*)
chord_body : chord_pitches

ReducedLyParser.**p_chord_body__chord_pitches__positive_leaf_duration**(*p*)
chord_body : chord_pitches positive_leaf_duration

ReducedLyParser.**p_chord_pitches__CARAT_L__pitches__CARAT_R**(*p*)
chord_pitches : CARAT_L pitches CARAT_R

ReducedLyParser.**p_commas__COMMA**(*p*)
commas : COMMA

ReducedLyParser.**p_commas__commas__commas**(*p*)
commas : commas COMMA

ReducedLyParser.**p_component__container**(*p*)
component : container

ReducedLyParser.**p_component__fixed_duration_container**(*p*)
component : fixed_duration_container

ReducedLyParser.**p_component__leaf**(*p*)
component : leaf

ReducedLyParser.**p_component__tuplet**(*p*)
component : tuplet

ReducedLyParser.**p_component_list__EMPTY**(*p*)
component_list :

ReducedLyParser.**p_component_list__component_list__component**(*p*)
component_list : component_list component

ReducedLyParser.**p_container__BRACE_L__component_list__BRACE_R**(*p*)
container : BRACE_L component_list BRACE_R

ReducedLyParser.**p_dots__EMPTY**(*p*)
dots :

ReducedLyParser.**p_dots__dots__DOT**(*p*)
dots : dots DOT

ReducedLyParser.**p_error**(*p*)

ReducedLyParser.**p_fixed_duration_container__BRACE_L__FRACTION__BRACE_R**(*p*)
fixed_duration_container : BRACE_L FRACTION BRACE_R

ReducedLyParser.**p_leaf__leaf_body__post_events**(*p*)
leaf : leaf_body post_events

ReducedLyParser.**p_leaf_body__chord_body**(*p*)
leaf_body : chord_body

ReducedLyParser.**p_leaf_body__note_body**(*p*)
leaf_body : note_body

ReducedLyParser.**p_leaf_body__rest_body**(*p*)
leaf_body : rest_body

ReducedLyParser.**p_measure__PIPE__FRACTION__component_list__PIPE**(*p*)
measure : PIPE FRACTION component_list PIPE

ReducedLyParser.**p_negative_leaf_duration__INTEGER_N__dots**(*p*)
negative_leaf_duration : INTEGER_N dots

ReducedLyParser.**p_note_body__pitch**(*p*)
note_body : pitch

ReducedLyParser.**p_note_body__pitch__positive_leaf_duration**(*p*)
note_body : pitch positive_leaf_duration

ReducedLyParser.**p_note_body__positive_leaf_duration**(*p*)
note_body : positive_leaf_duration

ReducedLyParser.**p_pitch__PITCHNAME**(*p*)
pitch : PITCHNAME

ReducedLyParser.**p_pitch__PITCHNAME__apostrophes**(*p*)
pitch : PITCHNAME apostrophes

ReducedLyParser.**p_pitch__PITCHNAME__commas**(*p*)
pitch : PITCHNAME commas

ReducedLyParser.**p_pitches__pitch**(*p*)
pitches : pitch

ReducedLyParser.**p_pitches__pitches__pitch**(*p*)
pitches : pitches pitch

ReducedLyParser.**p_positive_leaf_duration__INTEGER_P__dots**(*p*)
positive_leaf_duration : INTEGER_P dots

ReducedLyParser.**p_post_event__beam**(*p*)
post_event : beam

ReducedLyParser.**p_post_event__slur**(*p*)
post_event : slur

ReducedLyParser.**p_post_event__tie**(*p*)
post_event : tie

ReducedLyParser.**p_post_events__EMPTY**(*p*)
post_events :

ReducedLyParser.**p_post_events__post_events__post_event**(*p*)
post_events : post_events post_event

ReducedLyParser.**p_rest_body__RESTNAME**(*p*)
rest_body : RESTNAME

ReducedLyParser.**p_rest_body__RESTNAME__positive_leaf_duration**(*p*)
rest_body : RESTNAME positive_leaf_duration

ReducedLyParser.**p_rest_body__negative_leaf_duration**(*p*)
rest_body : negative_leaf_duration

ReducedLyParser.**p_slur__PAREN_L**(*p*)
slur : PAREN_L

ReducedLyParser.**p_slur__PAREN_R**(*p*)
slur : PAREN_R

ReducedLyParser.**p_start__EMPTY**(*p*)
start :

ReducedLyParser.**p_start__start__component**(*p*)
start : start component

ReducedLyParser.**p_start__start__measure**(*p*)
start : start measure

ReducedLyParser.**p_tie__TILDE**(*p*)
tie : TILDE

ReducedLyParser.**p_tuplet__FRACTION__container**(*p*)
tuplet : FRACTION container

ReducedLyParser.**t_FRACTION**(*t*)
([1-9]d*/[1-9]d*)

ReducedLyParser.**t_INTEGER_N**(*t*)
(-[1-9]d*)

ReducedLyParser.**t_INTEGER_P**(*t*)
([1-9]d*)

ReducedLyParser.**t_PITCHNAME**(*t*)
[a-g](fflsslflsltqfltqslqslqf)?

ReducedLyParser.**t_error**(*t*)

ReducedLyParser.**t_newline**(*t*)
n+

ReducedLyParser.**tokenize**(*input_string*)
Tokenize *input string* and print results.

Special methods

ReducedLyParser.**__call__**(*input_string*)
Parse *input_string* and return result.

ReducedLyParser.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

ReducedLyParser.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`ReducedLyParser.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`ReducedLyParser.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`ReducedLyParser.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

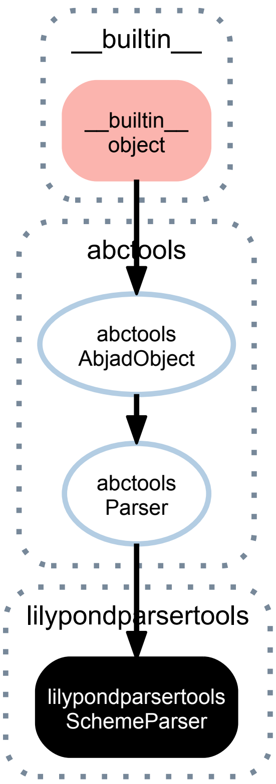
`ReducedLyParser.__ne__(expr)`
Defined equal to the opposite of equality.

Return boolean.

`ReducedLyParser.__repr__()`
Interpreter representation of Abjad object.

Return string.

62.1.9 lilypondparsertools.SchemeParser



class `lilypondparsertools.SchemeParser(debug=False)`
SchemeParser mimics how LilyPond's embedded Scheme parser behaves.

It parses a single Scheme expression and then stops, by raising a *SchemeParserFinishedException*.

The parsed expression and its exact length in characters are cached on the *SchemeParser* instance.

It is intended to be used only in conjunction with *LilyPondParser*.

Returns *SchemeParser* instance.

Read-only properties

SchemeParser.**debug**
True if the parser runs in debugging mode.

SchemeParser.**lexer**
The parser's PLY Lexer instance.

SchemeParser.**lexer_rules_object**

SchemeParser.**logger**
The parser's Logger instance.

SchemeParser.**logger_path**
The output path for the parser's logfile.

SchemeParser.**output_path**
The output path for files associated with the parser.

SchemeParser.**parser**
The parser's PLY LRParser instance.

SchemeParser.**parser_rules_object**

SchemeParser.**pickle_path**
The output path for the parser's pickled parsing tables.

SchemeParser.**storage_format**
Storage format of Abjad object.

Return string.

Methods

SchemeParser.**p_boolean__BOOLEAN**(*p*)
boolean : BOOLEAN

SchemeParser.**p_constant__boolean**(*p*)
constant : boolean

SchemeParser.**p_constant__number**(*p*)
constant : number

SchemeParser.**p_constant__string**(*p*)
constant : string

SchemeParser.**p_data__EMPTY**(*p*)
data :

SchemeParser.**p_data__data__datum**(*p*)
data : data datum

SchemeParser.**p_datum__constant**(*p*)
datum : constant

SchemeParser.**p_datum__list**(*p*)
datum : list

SchemeParser.**p_datum__symbol**(*p*)
datum : symbol

SchemeParser.**p_datum__vector**(*p*)
datum : vector

SchemeParser.**p_error**(*p*)

SchemeParser.**p_expression__QUOTE__datum**(*p*)
expression : QUOTE datum

SchemeParser.**p_expression__constant**(*p*)
expression : constant

SchemeParser.**p_expression__variable**(*p*)
expression : variable

SchemeParser.**p_form__expression**(*p*)
form : expression

SchemeParser.**p_forms__EMPTY**(*p*)
forms :

SchemeParser.**p_forms__forms__form**(*p*)
forms : forms form

SchemeParser.**p_list__L_PAREN__data__R_PAREN**(*p*)
list : L_PAREN data R_PAREN

SchemeParser.**p_list__L_PAREN__data__datum__PERIOD__datum__R_PAREN**(*p*)
list : L_PAREN data datum PERIOD datum R_PAREN

SchemeParser.**p_number__DECIMAL**(*p*)
number : DECIMAL

SchemeParser.**p_number__HEXADECIMAL**(*p*)
number : HEXADECIMAL

SchemeParser.**p_number__INTEGER**(*p*)
number : INTEGER

SchemeParser.**p_program__forms**(*p*)
program : forms

SchemeParser.**p_string__STRING**(*p*)
string : STRING

SchemeParser.**p_symbol__IDENTIFIER**(*p*)
symbol : IDENTIFIER

SchemeParser.**p_variable__IDENTIFIER**(*p*)
variable : IDENTIFIER

SchemeParser.**p_vector__HASH__L_PAREN__data__R_PAREN**(*p*)
vector : HASH L_PAREN data R_PAREN

SchemeParser.**t_BOOLEAN**(*t*)
#(T|F|t|f)

SchemeParser.**t_DECIMAL**(*t*)
(((-?[0-9]+).[0-9]*)|(-?.[0-9]+))

SchemeParser.**t_HASH**(*t*)
#

SchemeParser.**t_HEXADECIMAL**(*t*)
(X|x)[A-Fa-f0-9]+

SchemeParser.**t_IDENTIFIER**(*t*)
([A-Za-z!$%&*/<>?~_^:=][A-Za-z0-9!$%&*/<>?~_^:=.+-]*|[+-]|...)

SchemeParser.**t_INTEGER**(*t*)
(-?[0-9]+)

SchemeParser.**t_L_PAREN**(*t*)
(

SchemeParser.**t_R_PAREN**(*t*)
)

SchemeParser.**t_anything**(*t*)
.

SchemeParser.**t_error**(*t*)

SchemeParser.**t_newline**(*t*)
n+

SchemeParser.**t_quote**(*t*)
“

SchemeParser.**t_quote_440**(*t*)
\[nt\"']

SchemeParser.**t_quote_443**(*t*)
[^\"“]+

SchemeParser.**t_quote_446**(*t*)
“

SchemeParser.**t_quote_456**(*t*)
.

SchemeParser.**t_quote_error**(*t*)

SchemeParser.**t_whitespace**(*t*)
[tr]+

SchemeParser.**tokenize**(*input_string*)
Tokenize *input string* and print results.

Special methods

SchemeParser.**__call__**(*input_string*)
Parse *input_string* and return result.

SchemeParser.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

SchemeParser.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeParser.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

SchemeParser.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SchemeParser.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

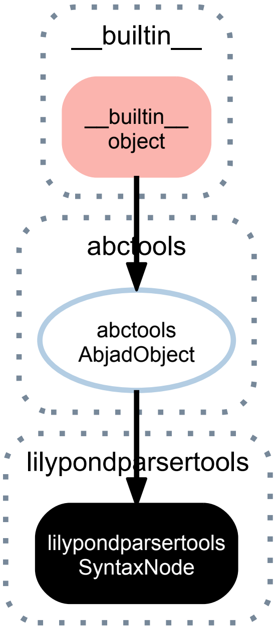
SchemeParser.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SchemeParser.**__repr__**()
Interpreter representation of Abjad object.

Return string.

62.1.10 lilypondparsertools.SyntaxNode



class `lilypondparsertools.`**`SyntaxNode`**(*type*, *value*=[ ])

A node in an abstract syntax tree (AST).

Not composer-safe.

Used internally by LilyPondParser.

Read-only properties

`SyntaxNode.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

`SyntaxNode.`**`__eq__`**(*expr*)

True when `id(self)` equals `id(expr)`.

Return boolean.

`SyntaxNode.`**`__ge__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

`SyntaxNode.`**`__getitem__`**(*item*)

`SyntaxNode.`**`__gt__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception

`SyntaxNode.`**`__le__`**(*expr*)

Abjad objects by default do not implement this method.

Raise exception.

SyntaxNode.**__len__**()

SyntaxNode.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

SyntaxNode.**__ne__**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

SyntaxNode.**__repr__**()

SyntaxNode.**__str__**()

62.2 Functions

62.2.1 lilypondparsertools.lilypond_enharmonic_transpose

lilypondparsertools.**lilypond_enharmonic_transpose**(*pitch_a*, *pitch_b*, *pitch_c*)
Transpose *pitch_c* by the distance between *pitch_b* and *pitch_a*.

This function was reverse-engineered from LilyPond's source code.

Return NamedChromaticPitch.

62.2.2 lilypondparsertools.list_known_contexts

lilypondparsertools.**list_known_contexts**()
List all LilyPond contexts recognized by `LilyPondParser`:

```
>>> for x in lilypondparsertools.list_known_contexts():
...     print x
...
ChoirStaff
ChordNames
CueVoice
Devnull
DrumStaff
DrumVoice
Dynamics
FiguredBass
FretBoards
Global
GrandStaff
GregorianTranscriptionStaff
GregorianTranscriptionVoice
KievanStaff
KievanVoice
Lyrics
MensuralStaff
MensuralVoice
NoteNames
PetrucciStaff
PetrucciVoice
PianoStaff
RhythmicStaff
Score
Staff
StaffGroup
TabStaff
TabVoice
VaticanaStaff
VaticanaVoice
Voice
```

Return list.

62.2.3 lilypondparsertools.list_known_grobs

lilypondparsertools.**list_known_grobs**()

List all LilyPond grobs recognized by `LilyPondParser`:

```
>>> for x in lilypondparsertools.list_known_grobs():
...     print x
...
Accidental
AccidentalCautionary
AccidentalPlacement
AccidentalSuggestion
Ambitus
AmbitusAccidental
AmbitusLine
AmbitusNoteHead
Arpeggio
BalloonTextItem
BarLine
BarNumber
BassFigure
BassFigureAlignment
BassFigureAlignmentPositioning
BassFigureBracket
BassFigureContinuation
BassFigureLine
Beam
BendAfter
BreakAlignGroup
BreakAlignment
BreathingSign
ChordName
Clef
ClusterSpanner
ClusterSpannerBeacon
CombineTextScript
CueClef
CueEndClef
Custos
DotColumn
Dots
DoublePercentRepeat
DoublePercentRepeatCounter
DoubleRepeatSlash
DynamicLineSpanner
DynamicText
DynamicTextSpanner
Episema
Fingering
FingeringColumn
Flag
FootnoteItem
FootnoteSpanner
FretBoard
Glissando
GraceSpacing
GridLine
GridPoint
Hairpin
HorizontalBracket
InstrumentName
InstrumentSwitch
KeyCancellation
KeySignature
LaissezVibrerTie
LaissezVibrerTieColumn
LedgerLineSpanner
LeftEdge
LigatureBracket
```

```
    LyricExtender
    LyricHyphen
    LyricSpace
    LyricText
    MeasureCounter
    MeasureGrouping
    MelodyItem
    MensuralLigature
    MetronomeMark
    MultiMeasureRest
    MultiMeasureRestNumber
    MultiMeasureRestText
    NonMusicalPaperColumn
    NoteCollision
    NoteColumn
    NoteHead
    NoteName
    NoteSpacing
    OctavateEight
    OttavaBracket
    PaperColumn
    ParenthesesItem
    PercentRepeat
    PercentRepeatCounter
    PhrasingSlur
    PianoPedalBracket
    RehearsalMark
    RepeatSlash
    RepeatTie
    RepeatTieColumn
    Rest
    RestCollision
    Script
    ScriptColumn
    ScriptRow
    Slur
    SostenutoPedal
    SostenutoPedalLineSpanner
    SpacingSpanner
    SpanBar
    SpanBarStub
    StaffGrouper
    StaffSpacing
    StaffSymbol
    StanzaNumber
    Stem
    StemStub
    StemTremolo
    StringNumber
    StrokeFinger
    SustainPedal
    SustainPedalLineSpanner
    System
    SystemStartBar
    SystemStartBrace
    SystemStartBracket
    SystemStartSquare
    TabNoteHead
    TextScript
    TextSpanner
    Tie
    TieColumn
    TimeSignature
    TrillPitchAccidental
    TrillPitchGroup
    TrillPitchHead
    TrillSpanner
    TupletBracket
    TupletNumber
    UnaCordaPedal
    UnaCordaPedalLineSpanner
    VaticanaLigature
    VerticalAlignment
```

```
VerticalAxisGroup
VoiceFollower
VoltaBracket
VoltaBracketSpanner
```

Return tuple.

62.2.4 lilypondparsertools.list_known_languages

lilypondparsertools.**list_known_languages**()

List all note-input languages recognized by `LilyPondParser`:

```
>>> for x in lilypondparsertools.list_known_languages():
...     print x
...
catalan
deutsch
english
espanol
español
français
italiano
nederlands
norsk
portugues
suomi
svenska
vlaams
```

Return list.

62.2.5 lilypondparsertools.list_known_markup_functions

lilypondparsertools.**list_known_markup_functions**()

List all markup functions recognized by `LilyPondParser`:

```
>>> for x in lilypondparsertools.list_known_markup_functions():
...     print x
...
abs-fontsize
arrow-head
auto-footnote
backslashed-digit
beam
bold
box
bracket
caps
center-align
center-column
char
circle
column
column-lines
combine
concat
customTabClef
dir-column
doubleflat
doublesharp
draw-circle
draw-hline
draw-line
dynamic
epsfile
eyeglasses
fill-line
```

```
    fill-with-pattern
    filled-box
    finger
    flat
    fontCaps
    fontsize
    footnote
    fraction
    fret-diagram
    fret-diagram-terse
    fret-diagram-verbose
    fromproperty
    general-align
    halign
    harp-pedal
    hbracket
    hcenter-in
    hspace
    huge
    italic
    justified-lines
    justify
    justify-field
    justify-string
    large
    larger
    left-align
    left-brace
    left-column
    line
    lookup
    lower
    magnify
    markalphabet
    markletter
    medium
    musicglyph
    natural
    normal-size-sub
    normal-size-super
    normal-text
    normalsize
    note
    note-by-number
    null
    number
    on-the-fly
    override
    override-lines
    pad
    pad-around
    pad-to-box
    pad-x
    page-link
    page-ref
    parenthesize
    path
    pattern
    postscript
    property-recursive
    put-adjacent
    raise
    replace
    rest
    rest-by-number
    right-align
    right-brace
    right-column
    roman
    rotate
    rounded-box
    sans
    scale
```

```
score
semiflat
semisharp
sesquiflat
sesquisharp
sharp
simple
slashed-digit
small
smallCaps
smaller
stencil
strut
sub
super
table-of-contents
teeny
text
tied-lyric
tiny
translate
translate-scaled
transparent
triangle
typewriter
underline
upright
vcenter
verbatim-file
vspace
whiteout
with-color
with-dimensions
with-link
with-url
woodwind-diagram
wordwrap
wordwrap-field
wordwrap-internal
wordwrap-lines
wordwrap-string
wordwrap-string-internal
```

Return list.

62.2.6 lilypondparsertools.list_known_music_functions

lilypondparsertools.**list_known_music_functions**()

List all music functions recognized by `LilyPondParser`:

```
>>> for x in lilypondparsertools.list_known_music_functions():
...     print x
...
acciaccatura
appoggiatura
bar
breathe
clef
grace
key
language
makeClusters
mark
relative
skip
time
times
transpose
```

Return list.

62.2.7 lilypondparsertools.parse_reduced_ly_syntax

lilypondparsertools.**parse_reduced_ly_syntax**(*string*)

Parse the reduced LilyPond rhythmic syntax:

```
>>> string = '4 -4. 8.. 5/3 { } 4'
>>> result = lilypondparsertools.parse_reduced_ly_syntax(string)
```

```
>>> for x in result:
...     x
...
Note("c'4")
Rest('r4.')
Note("c'8..")
Tuplet(5/3, [])
Note("c'4")
```

Return list.

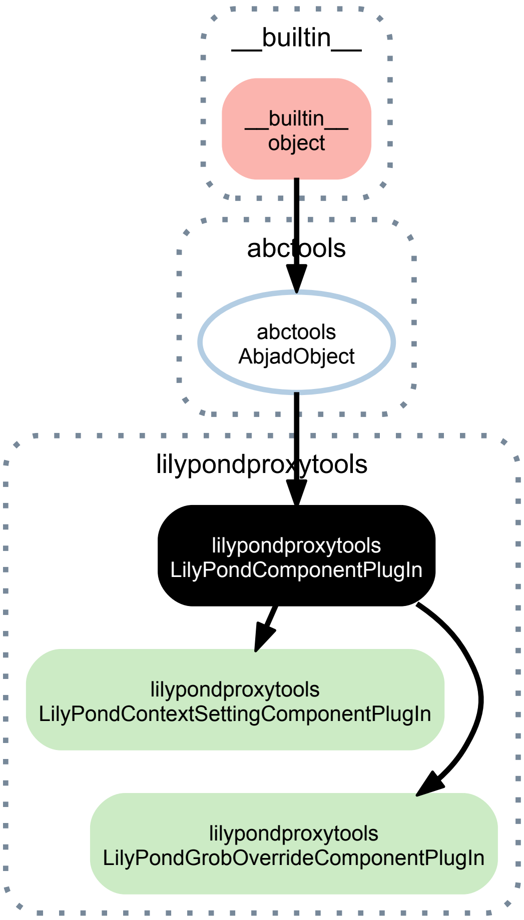
CHAPTER

SIXTYTHREE

LILYPONDPROXYTOOLS

63.1 Concrete Classes

63.1.1 lilypondproxytools.LilyPondComponentPlugIn



class `lilypondproxytools.`**`LilyPondComponentPlugIn`**(***kwargs*)

New in version 2.0. Shared LilyPond grob proxy and LilyPond context proxy functionality.

Read-only properties

`LilyPondComponentPlugIn.`**`storage_format`**

Storage format of Abjad object.

Return string.

Special methods

LilyPondComponentPlugIn.**__eq__**(*arg*)

LilyPondComponentPlugIn.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondComponentPlugIn.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondComponentPlugIn.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

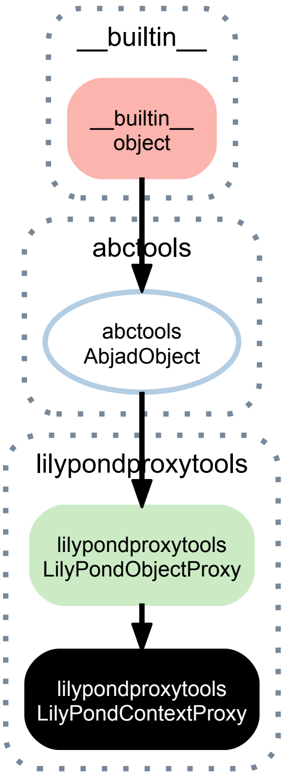
LilyPondComponentPlugIn.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondComponentPlugIn.**__ne__**(*arg*)

LilyPondComponentPlugIn.**__repr__**()

63.1.2 lilypondproxytools.LilyPondContextProxy



class lilypondproxytools.**LilyPondContextProxy**(***kwargs*)
New in version 2.0. LilyPond context proxy.

Read-only properties

LilyPondContextProxy.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

LilyPondContextProxy.**__copy__**()

LilyPondContextProxy.**__eq__**(*arg*)

LilyPondContextProxy.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondContextProxy.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondContextProxy.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

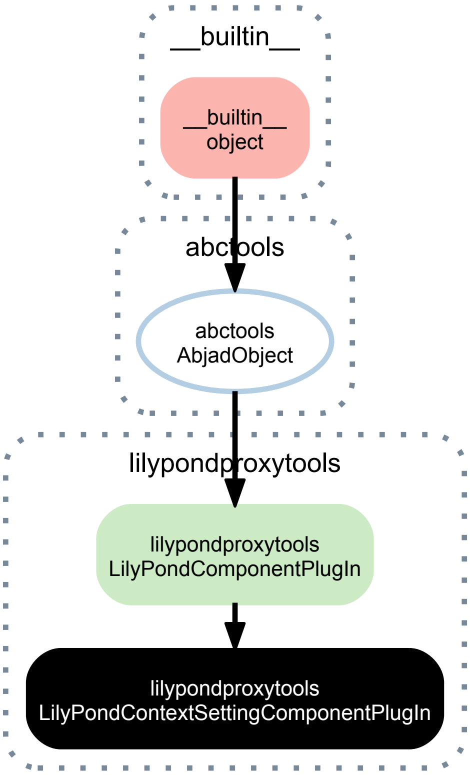
LilyPondContextProxy.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondContextProxy.**__ne__**(*arg*)

LilyPondContextProxy.**__repr__**()

63.1.3 lilypondproxytools.LilyPondContextSettingComponentPlugIn



class `lilypondproxytools.`**`LilyPondContextSettingComponentPlugIn`**(***kwargs*)
New in version 2.0. LilyPond context setting namespace.

Read-only properties

`LilyPondContextSettingComponentPlugIn.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`LilyPondContextSettingComponentPlugIn.`**`__eq__`**(*arg*)

`LilyPondContextSettingComponentPlugIn.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondContextSettingComponentPlugIn.`**`__getattr__`**(*name*)

`LilyPondContextSettingComponentPlugIn.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

`LilyPondContextSettingComponentPlugIn.`**`__le__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

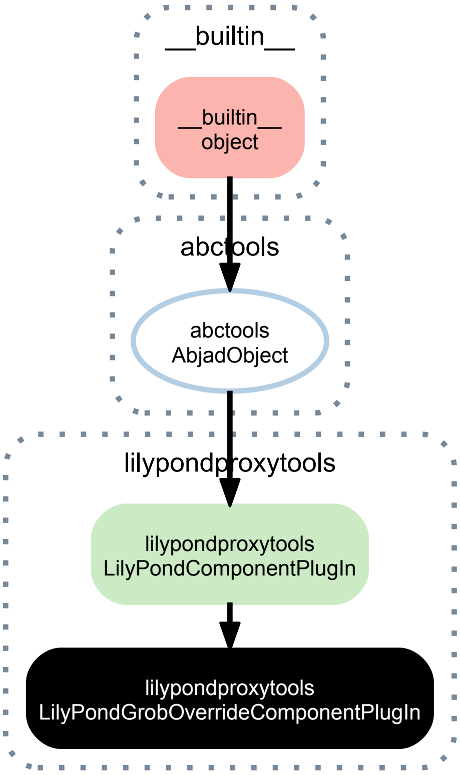
`LilyPondContextSettingComponentPlugIn.`**`__lt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondContextSettingComponentPlugIn.**__ne__**(*arg*)

LilyPondContextSettingComponentPlugIn.**__repr__**()

63.1.4 lilypondproxytools.LilyPondGrobOverrideComponentPlugIn



class lilypondproxytools.**LilyPondGrobOverrideComponentPlugIn**(***kwargs*)
New in version 2.0. LilyPond grob override component plug-in.

Read-only properties

LilyPondGrobOverrideComponentPlugIn.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

LilyPondGrobOverrideComponentPlugIn.**__eq__**(*arg*)

LilyPondGrobOverrideComponentPlugIn.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondGrobOverrideComponentPlugIn.**__getattr__**(*name*)

LilyPondGrobOverrideComponentPlugIn.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondGrobOverrideComponentPlugIn.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondGrobOverrideComponentPlugIn.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

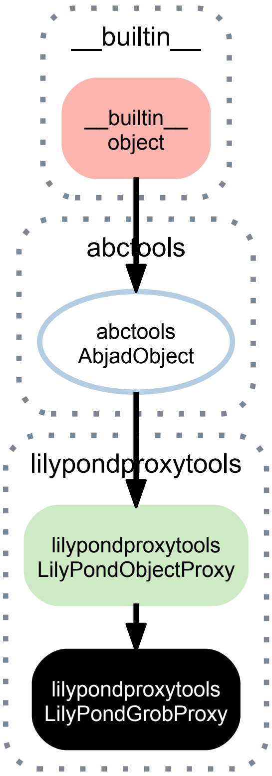
Raise exception.

LilyPondGrobOverrideComponentPlugIn.**__ne__**(*arg*)

LilyPondGrobOverrideComponentPlugIn.**__repr__**()

LilyPondGrobOverrideComponentPlugIn.**__setattr__**(*attr*, *value*)

63.1.5 lilypondproxytools.LilyPondGrobProxy



class lilypondproxytools.**LilyPondGrobProxy**(***kwargs*)
New in version 2.0. LilyPond grob proxy.

Read-only properties

LilyPondGrobProxy.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

LilyPondGrobProxy.**__copy__**()

LilyPondGrobProxy.**__eq__**(*arg*)

LilyPondGrobProxy.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondGrobProxy.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondGrobProxy.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

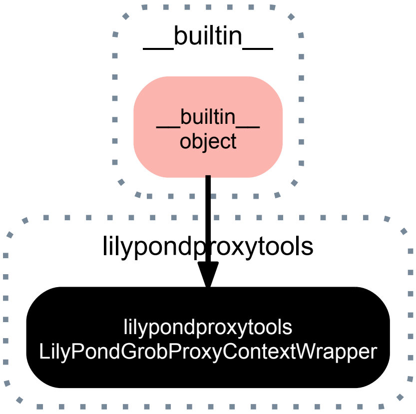
LilyPondGrobProxy.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondGrobProxy.**__ne__**(*arg*)

LilyPondGrobProxy.**__repr__**()

63.1.6 lilypondproxytools.LilyPondGrobProxyContextWrapper



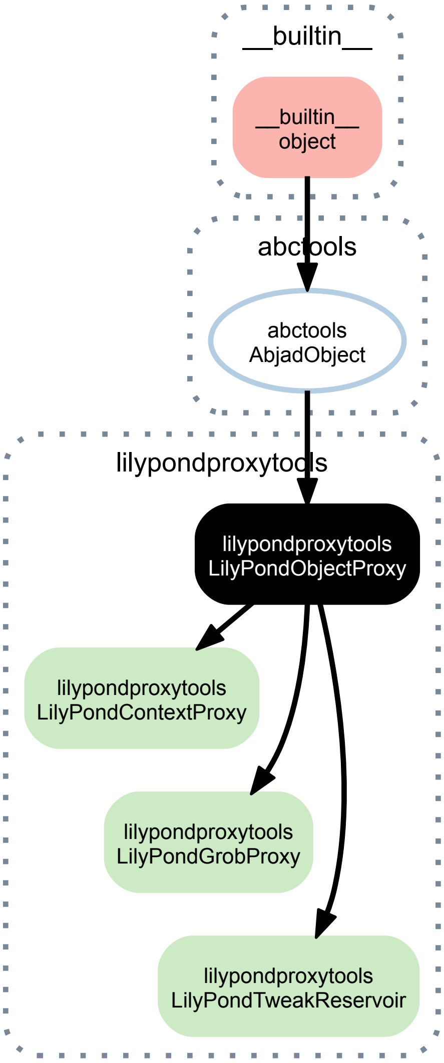
class lilypondproxytools.**LilyPondGrobProxyContextWrapper**
New in version 2.0. Context wrapper for LilyPond grob overrides.

Special methods

LilyPondGrobProxyContextWrapper.**__getattr__**(*name*)

LilyPondGrobProxyContextWrapper.**__repr__**()

63.1.7 lilypondproxytools.LilyPondObjectProxy



class `lilypondproxytools.`**`LilyPondObjectProxy`**(***kwargs*)
New in version 2.0. Shared LilyPond grob proxy and LilyPond context proxy functionality.

Read-only properties

`LilyPondObjectProxy.`**`storage_format`**
Storage format of Abjad object.

Return string.

Special methods

`LilyPondObjectProxy.`**`__copy__`**()

`LilyPondObjectProxy.`**`__eq__`**(*arg*)

`LilyPondObjectProxy.`**`__ge__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondObjectProxy.`**`__gt__`**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

LilyPondObjectProxy.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

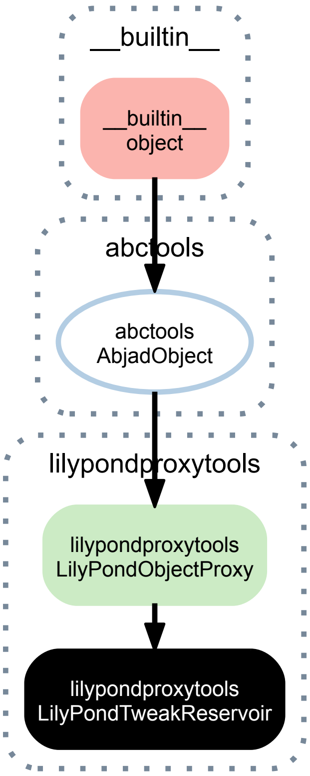
LilyPondObjectProxy.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

LilyPondObjectProxy.**__ne__**(*arg*)

LilyPondObjectProxy.**__repr__**()

63.1.8 lilypondproxytools.LilyPondTweakReservoir



class lilypondproxytools.**LilyPondTweakReservoir**(***kwargs*)
New in version 2.0. LilyPond tweak reservoir.

Read-only properties

LilyPondTweakReservoir.**storage_format**
Storage format of Abjad object.

Return string.

Special methods

LilyPondTweakReservoir.**__copy__**()

LilyPondTweakReservoir.**__eq__**(*arg*)

`LilyPondTweakReservoir.__ge__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondTweakReservoir.__gt__(expr)`
Abjad objects by default do not implement this method.

Raise exception

`LilyPondTweakReservoir.__le__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondTweakReservoir.__lt__(expr)`
Abjad objects by default do not implement this method.

Raise exception.

`LilyPondTweakReservoir.__ne__(arg)`

`LilyPondTweakReservoir.__repr__()`

CHAPTER

SIXTYFOUR

OFFSETTOOLS

64.1 Functions

64.1.1 offsettools.update_offset_values_of_component

offsettools.**update_offset_values_of_component**(*component*)

New in version 2.9. Update prolated offset values of *component*.

64.1.2 offsettools.update_offset_values_of_component_in_seconds

offsettools.**update_offset_values_of_component_in_seconds**(*component*)

New in version 2.9. Update offset values of *component* in seconds.

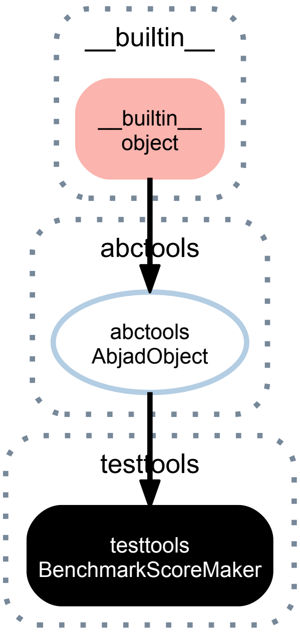
CHAPTER

SIXTYFIVE

TESTTOOLS

65.1 Concrete Classes

65.1.1 testtools.BenchmarkScoreMaker



class testtools.**BenchmarkScoreMaker**

New in version 2.12. Benchmark score maker:

```
>>> benchmark_score_maker = testtools.BenchmarkScoreMaker()
```

```
>>> benchmark_score_maker
BenchmarkScoreMaker()
```

Use to instantiate scores for benchmark testing.

Read-only properties

BenchmarkScoreMaker.**storage_format**

Storage format of Abjad object.

Return string.

Methods

`BenchmarkScoreMaker.`**`make_bound_hairpin_score_01`**`()`

Make 200-note voice with p-to-f bound crescendo spanner on every 4 notes.

```
2.12 (r9726) initialization:        279,448 function calls

2.12 (r9726) LilyPond format:       124,517 function calls
```

`BenchmarkScoreMaker.`**`make_bound_hairpin_score_02`**`()`

Make 200-note voice with p-to-f bound crescendo spanner on every 20 notes.

```
2.12 (r9726) initialization:        268,845 function calls

2.12 (r9726) LilyPond format:       117,846 function calls
```

`BenchmarkScoreMaker.`**`make_bound_hairpin_score_03`**`()`

Make 200-note voice with p-to-f bound crescendo spanner on every 100 notes.

```
2.12 (r9726) initialization:        267,417 function calls

2.12 (r9726) LilyPond format:       116,534 function calls
```

`BenchmarkScoreMaker.`**`make_context_mark_score_01`**`()`

Make 200-note voice with dynamic mark on every 20th note:

```
2.12 (r9704) initialization:        630,433 function calls
2.12 (r9710) initialization:        235,120 function calls
2.12 r(9726) initialization:        235,126 function calls

2.12 (r9704) LilyPond format:       136,637 function calls
2.12 (r9710) LilyPond format:        82,730 function calls
2.12 (r9726) LilyPond format:        88,382 function calls
```

`BenchmarkScoreMaker.`**`make_context_mark_score_02`**`()`

Make 200-note staff with dynamic mark on every 4th note.

```
2.12 (r9704) initialization:      4,632,761 function calls
2.12 (r9710) initialization:        327,280 function calls
2.12 (r9726) initialization:        325,371 function calls

2.12 (r9704) LilyPond format:       220,277 function calls
2.12 (r9710) LilyPond format:        84,530 function calls
2.12 (r9726) LilyPond format:        90,056 function calls
```

`BenchmarkScoreMaker.`**`make_context_mark_score_03`**`()`

Make 200-note staff with dynamic mark on every note.

```
2.12 (r9704) initialization:     53,450,195 function calls (!!)
2.12 (r9710) initialization:      2,124,500 function calls
2.12 (r9724) initialization:      2,122,591 function calls

2.12 (r9704) LilyPond format:       533,927 function calls
2.12 (r9710) LilyPond format:        91,280 function calls
2.12 (r9724) LilyPond format:        96,806 function calls
```

`BenchmarkScoreMaker.`**`make_hairpin_score_01`**`()`

Make 200-note voice with crescendo spanner on every 4 notes.

```
2.12 (r9726) initialization:        248,502 function calls
2.12 (r9728) initialization:        248,502 function calls

2.12 (r9726) LilyPond format:       138,313 function calls
2.12 (r9728) LilyPond format:       134,563 function calls
```

`BenchmarkScoreMaker.`**`make_hairpin_score_02`**`()`

Make 200-note voice with crescendo spanner on every 20 notes.

```
2.12 (r9726) initialization:        248,687 function calls
2.12 (r9728) initialization:        248,687 function calls

2.12 (r9726) LilyPond format:       134,586 function calls
2.12 (r9728) LilyPond format:       129,836 function calls
```

`BenchmarkScoreMaker.`**`make_hairpin_score_03`**`()`

Make 200-note voice with crescendo spanner on every 100 notes.

```
2.12 (r9726) initialization:        249,363 function calls
2.12 (r9726) initialization:        249,363 function calls

2.12 (r9726) LilyPond format:       133,898 function calls
2.12 (r9728) LilyPond format:       128,948 function calls
```

`BenchmarkScoreMaker.`**`make_score_00`**`()`

Make 200-note voice (with nothing else).

```
2.12 (r9710) initialization:        156,821 function calls
2.12 (r9726) initialization:        156,827 function calls

2.12 (r9703) LilyPond format:        99,127 function calls
2.12 (r9710) LilyPond format:       100,126 function calls
2.12 (r9726) LilyPond format:       105,778 function calls
```

`BenchmarkScoreMaker.`**`make_spanner_score_01`**`()`

Make 200-note voice with durated complex beam spanner on every 4 notes.

```
2.12 (r9710) initialization:        248,654 function calls
2.12 (r9724) initialization:        248,660 function calls

2.12 (r9703) LilyPond format:       425,848 function calls
2.12 (r9710) LilyPond format:       426,652 function calls
2.12 (r9724) LilyPond format:       441,884 function calls
```

`BenchmarkScoreMaker.`**`make_spanner_score_02`**`()`

Make 200-note voice with durated complex beam spanner on every 20 notes.

```
2.12 (r9710) initialization:        250,954 function calls
2.12 (r9724) initialization:        248,717 function calls

2.12 (r9703) LilyPond format:       495,768 function calls
2.12 (r9710) LilyPond format:       496,572 function calls
2.12 (r9724) LilyPond format:       511,471 function calls
```

`BenchmarkScoreMaker.`**`make_spanner_score_03`**`()`

Make 200-note voice with durated complex beam spanner on every 100 notes.

```
2.12 (r9710) initialization:        251,606 function calls
2.12 (r9724) initialization:        249,369 function calls

2.12 (r9703) LilyPond format:       509,752 function calls
2.12 (r9710) LilyPond format:       510,556 function calls
2.12 (r9724) LilyPond format:       525,463 function calls
```

`BenchmarkScoreMaker.`**`make_spanner_score_04`**`()`

Make 200-note voice with slur spanner on every 4 notes.

```
2.12 (r9724) initialization:        245,683 function calls

2.12 (r9703) LilyPond format:       125,577 function calls
2.12 (r9724) LilyPond format:       111,341 function calls
```

BenchmarkScoreMaker.**make_spanner_score_05**()
Make 200-note voice with slur spanner on every 20 notes.

```
2.12 (r9724) initialization:        248,567 function calls

2.12 (r9703) LilyPond format:       122,177 function calls
2.12 (r9724) LilyPond format:       107,486 function calls
```

BenchmarkScoreMaker.**make_spanner_score_06**()
Make 200-note voice with slur spanner on every 100 notes.

```
2.12 (r9724) initialization:        249,339 function calls

2.12 (r9703) LilyPond format:       121,497 function calls
2.12 (r9724) LilyPond format:       106,718 function calls
```

BenchmarkScoreMaker.**make_spanner_score_07**()
Make 200-note voice with (vanilla) beam spanner on every 4 notes.

```
2.12 (r9724) initialization:        245,683 function calls

2.12 (r9703) LilyPond format:       125,577 function calls
2.12 (r9724) LilyPond format:       132,556 function calls
```

BenchmarkScoreMaker.**make_spanner_score_08**()
Make 200-note voice with (vanilla) beam spanner on every 20 notes.

```
2.12 (r9724) initialization:        248,567 function calls

2.12 (r9703) LilyPond format:       122,177 function calls
2.12 (r9724) LilyPond format:       129,166 function calls
```

BenchmarkScoreMaker.**make_spanner_score_09**()
Make 200-note voice with (vanilla) beam spanner on every 100 notes.

```
2.12 (r9724) initialization:        249,339 function calls

2.12 (r9703) LilyPond format:       121,497 function calls
2.12 (r9724) LilyPond format:       128,494 function calls
```

Special methods

BenchmarkScoreMaker.**__eq__**(*expr*)
True when `id(self)` equals `id(expr)`.

Return boolean.

BenchmarkScoreMaker.**__ge__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BenchmarkScoreMaker.**__gt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception

BenchmarkScoreMaker.**__le__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

BenchmarkScoreMaker.**__lt__**(*expr*)
Abjad objects by default do not implement this method.

Raise exception.

`BenchmarkScoreMaker.`**`__ne__`**(*expr*)
Defined equal to the opposite of equality.

Return boolean.

`BenchmarkScoreMaker.`**`__repr__`**()
Interpreter representation of Abjad object.

Return string.

65.2 Functions

65.2.1 testtools.apply_additional_layout

`testtools.`**`apply_additional_layout`**(*lilypond_file*)
Configure multiple-voice rhythmic staves in *lilypond_file*.

Operate in place and return none.

65.2.2 testtools.read_test_output

`testtools.`**`read_test_output`**(*full_file_name*, *current_function_name*)
Read test output.

65.2.3 testtools.write_test_output

`testtools.`**`write_test_output`**(*output*, *full_file_name*, *test_function_name*, *cache_ly=False*, *cache_pdf=False*, *go=False*, *render_pdf=False*)
Write test output.