auxjad Documentation

Release 0.7.0

Gilberto Agostinho

May 21, 2020

CONTENTS

GitHub | PyPI | Documentation | Issue Tracker | Travis-CI

`auxjad` is a library of auxiliary functions and classes for Abjad 3.1. All classes and functions have a __doc__ attribute with usage instructions.

Documentation is available at https://gilbertohasnofb.github.io/auxjad-docs/. A pdf version of the documentation is also available in the `/docs` directory of the repository.

Bugs can be reported through the project's Issue Tracker.

CHAPTER

ONE

INSTALLATION

The recommended way to install `auxjad` is via pip:

```
~$ pip install --user auxjad
```

Or if you are using virtual environments, simply use:

```
~$ pip install auxjad
```

You will also need to install Python 3.6 or higher, as well as Abjad and LilyPond.

1.1 Examples of usage

In this page there are two more complex examples showcasing the types of manipulation that `auxjad`'s functions and classes can accomplish. These examples do not make use of all functions and classes, nor does it show all the features of the used ones. Please refer to `auxjad`'s API in the left pane for the documentation of specific functions and classes.

1.1.1 Example 1

In this second example, we will use some of `auxjad`'s classes to manipulate some musical material using the looping and shuffling classes.

First, we start by importing both `abjad` and `auxjad`.

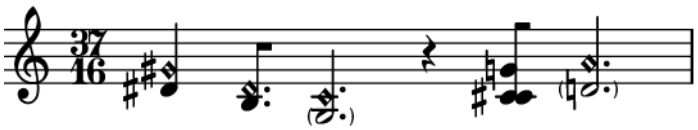
```
>>> import abjad
>>> import auxjad
```

Let's now create a container with some arbitrary material to be manipulated. Let's use the class `auxjad.ArtificialHarmonic` as well as some chords and rests.

```
>>> container = abjad.Staff([
...     auxjad.ArtificialHarmonic("<ds' gs'>4"),
...     auxjad.ArtificialHarmonic("<b ds'>8."),
...     auxjad.ArtificialHarmonic("<g c'>2.", is_parenthesized=True),
...     abjad.Rest("r4"),
...     abjad.Chord([0, 1, 7], (1, 8)),
...     auxjad.ArtificialHarmonic("<d' a'>2.", is_parenthesized=True),
... ])
```

Let's now add a time signature of the length of the container.

```
>>> container_length = abjad.inspect(container).duration()
>>> abjad.attach(abjad.TimeSignature(container_length), container[0])
>>> abjad.f(container)
\new Staff
{
    \time 37/16
    <
        ds'
        \tweak style #'harmonic
        gs'
    >4
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' cs' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
}
```



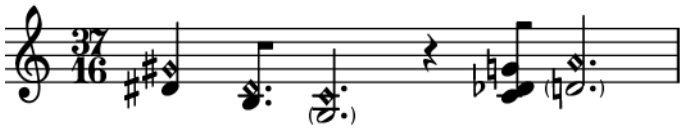
The spelling of the chord `<c' cs' g'>` could be improved. This can be done using either `auxjad.respell_chord` or `auxjad.respell_container`.

```
>>> auxjad.respell_container(container)
>>> abjad.f(container)
\new Staff
{
    \time 37/16
    <
        ds'
        \tweak style #'harmonic
        gs'
    >4
    <
        b
        \tweak style #'harmonic
        ds'
```

(continues on next page)

(continued from previous page)

```
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
}
```



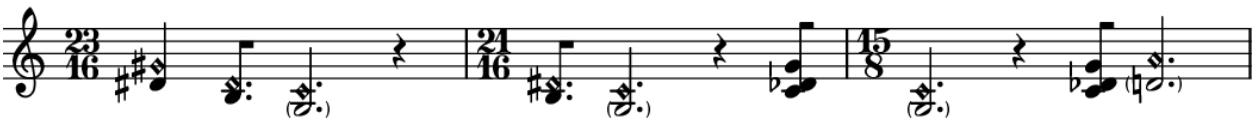
Let's now use this material as input for `auxjad.LoopByNotes`. This is one of the many loopers included in `auxjad`. It works by selecting groups of _n_ elements (given by the argument `window_size`). With `window_size` set to 4, this looper will first output the first four elements, then output elements 2 through 5, then 3 through 6, and so on.

```
>>> looper = auxjad.LoopByNotes(container, window_size=4)
>>> staff = abjad.Staff()
>>> for _ in range(3):
...     music = looper()
...     staff.append(music)
>>> abjad.f(staff)
\new Staff
{
    \time 23/16
    <
        ds'
        \tweak style #'harmonic
        gs'
    >4
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
```

(continues on next page)

(continued from previous page)

```
    r4
    \time 21/16
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    \time 15/8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
}
```



Let's now grab the last window output by the looper object above and use it as input for `auxjad.Shuffler`. This will randomly shuffles the leaves of the input container.

```
>>> container = abjad.Container(looper.current_window)
>>> shuffler = auxjad.Shuffler(container, omit_time_signatures=True)
>>> for _ in range(3):
...     music = shuffler()
...     staff.append(music)
>>> abjad.f(staff)
\new Staff
{
    \time 23/16
    <
        ds'
        \tweak style #'harmonic
        gs'
```

(continues on next page)

(continued from previous page)

```
    >4
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    \time 21/16
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    \time 15/8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
    r4
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    ~
    <
        \parenthesize
```

(continues on next page)

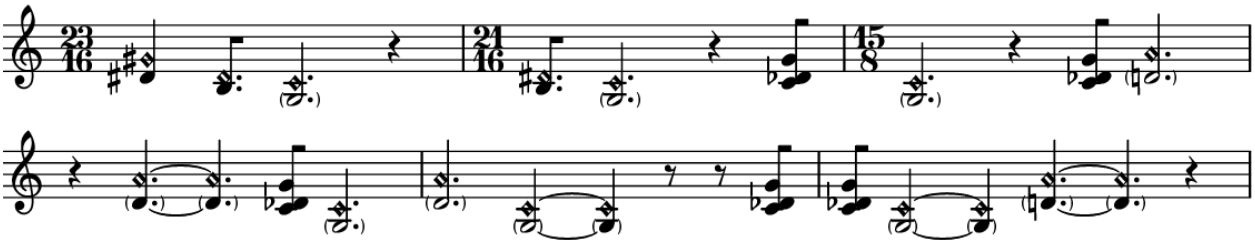
(continued from previous page)

```
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >4
    r8
    r8
    <c' df' g'>8
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >4
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
```

(continues on next page)

(continued from previous page)

```
        d'
        \tweak style #'harmonic
        a'
    >4.
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    r4
}
```



To finalise the score, let's add an initial dynamic to the first leaf of the staff.

```
>>> abjad.attach(abjad.Dynamic('ppp'), staff[0])
>>> abjad.f(staff)
\new Staff
{
    \time 23/16
    <
        ds'
        \tweak style #'harmonic
        gs'
    >4
    \ppp
    <
        b
        \tweak style #'harmonic
        ds'
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    \time 21/16
    <
        b
        \tweak style #'harmonic
        ds'
```

(continues on next page)

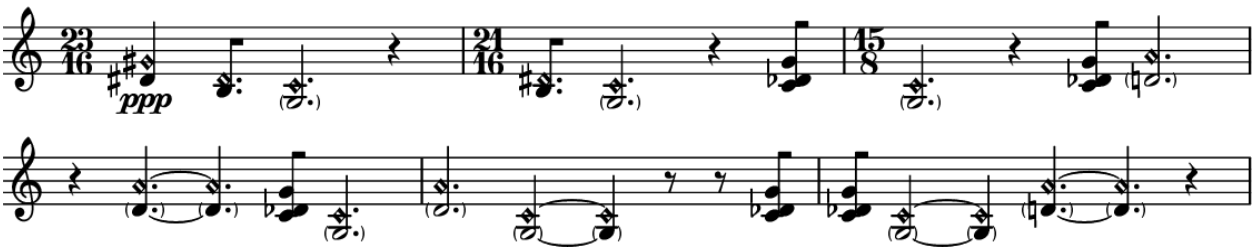
(continued from previous page)

```
    >8.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    \time 15/8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    r4
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >2.
    r4
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
```

(continues on next page)

(continued from previous page)

```
    >2.
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >4
    r8
    r8
    <c' df' g'>8
    <c' df' g'>8
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >2
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        g
        \tweak style #'harmonic
        c'
    >4
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    ~
    <
        \parenthesize
        \tweak ParenthesesItem.font-size #-4
        d'
        \tweak style #'harmonic
        a'
    >4.
    r4
}
```



1.1.2 Example 2

In this second example, we will use some of `auxjad`'s' classes to generate a container of randomly selected material, and then use this material as input for the looping and shuffling classes.

First, we start by importing both `abjad` and `auxjad`.

```
>>> import abjad
>>> import auxjad
```

Let's start by deciding what random selectors will be responsible for generating each parameter of our basic material. Let's use `auxjad.TenneySelector` for pitches, which is an implementation of Tenney's Dissonant Counterpoint Algorithm; at each call, this algorithm prioritises elements that haven't been select for the longest time. For the durations, dynamics, and articulations, the example will use `auxjad.CartographySelector`. Each element input into this type of selector has a probability of being selected which is dependent on its index. By default, the probability of consecutive elements decay with a rate of 0.75. For more information on both of these classes, check the `auxjad` API page (link in the left panel).

```
>>> pitch_selector = auxjad.TenneySelector([0, 7, 8, 2, 3, 10])
>>> duration_selector = auxjad.CartographySelector([(2, 8),
...                                                 (3, 8),
...                                                 (5, 8),
...                                                 ])
>>> dynamic_selector = auxjad.CartographySelector(['p', 'mp', 'mf', 'f'])
>>> articulation_selector = auxjad.CartographySelector([None, '-', '>'])
```

Let's now create eight random notes, each with four parameters randomly selected by the classes above.

```
>>> pitches = [pitch_selector() for _ in range(8)]
>>> durations = [duration_selector() for _ in range(8)]
>>> dynamics = [dynamic_selector() for _ in range(8)]
>>> articulations = [articulation_selector() for _ in range(8)]
```

With these lists of pitches, durations, dynamics, and articulations, we can now use `auxjad.LeafDynMaker` to create the individual abjad leaves for us.

```
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> container = abjad.Staff(notes)
```

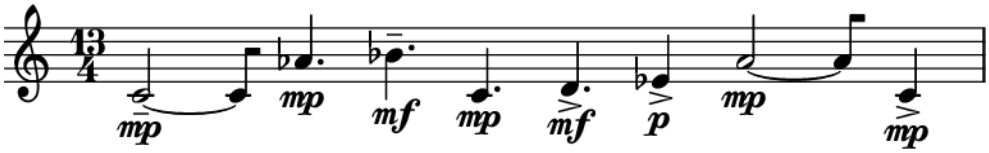
Let's now add a time signature of the length of the container.

```
>>> container_length = abjad.inspect(container).duration()
>>> abjad.attach(abjad.TimeSignature(container_length), container[0])
>>> abjad.f(container)
\new Staff
{
```

(continues on next page)

(continued from previous page)

```
    \time 13/4
    c'2
    \mp
    - \tenuto
    ~
    c'8
    af'4.
    \mp
    bf'4.
    \mf
    - \tenuto
    c'4.
    \mp
    d'4.
    \mf
    - \accent
    ef'4
    \p
    - \accent
    af'2
    \mp
    ~
    af'8
    c'4
    \mp
    - \accent
}
```



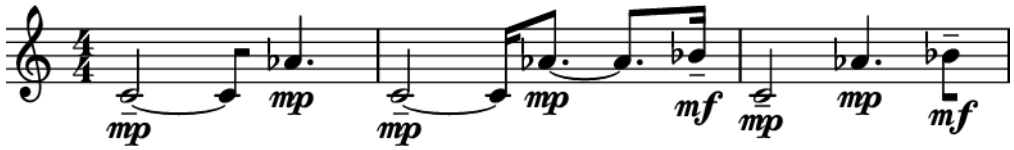
Let's now use `auxjad.LoopByWindow` to output loops of windows of the material. By default, this class uses a window size of a 4/4 measure, and each step forward has the size of a sixteenth-note. These parameters are all adjustable, please refer to this library's API for more information.

```
>>> looper = auxjad.LoopByWindow(container)
>>> staff = abjad.Staff()
>>> for _ in range(3):
>>>     music = looper()
>>>     staff.append(music)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'2
    \mp
    - \tenuto
    ~
    c'8
    af'4.
    \mp
    c'2
```

(continues on next page)

(continued from previous page)

```
    \mp
    - \tenuto
    ~
    c'16
    af'8.
    \mp
    ~
    af'8.
    bf'16
    \mf
    - \tenuto
    c'2
    \mp
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
}
```



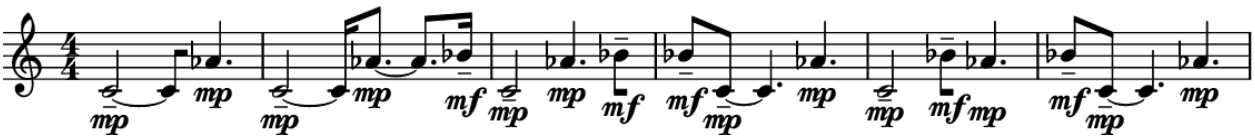
Let's now grab the last window output by the looper object above and use it as input for `auxjad.Shuffler`. This will randomly shuffles the leaves of the input container.

```
>>> container = abjad.Container(looper.current_window)
>>> shuffler = auxjad.Shuffler(container, omit_time_signatures=True)
>>> for _ in range(3):
>>>     music = shuffler()
>>>     staff.append(music)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'2
    \mp
    - \tenuto
    ~
    c'8
    af'4.
    \mp
    c'2
    \mp
    - \tenuto
    ~
    c'16
    af'8.
    \mp
    ~
    af'8.
    bf'16
```

(continues on next page)

(continued from previous page)

```
    \mf
    - \tenuto
    c'2
    \mp
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
    \mp
    c'2
    \mp
    - \tenuto
    bf'8
    \mf
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
    \mp
}
```



Let's use the last output of the shuffler above and feed it into a new looper. This time we will use a window of size 3/4.

```
>>> container = abjad.Container(shuffler.current_container)
>>> looper = auxjad.LoopByWindow(container,
...                              window_size=(3, 4),
...                              )
>>> for _ in range(3):
>>>     music = looper()
>>>     staff.append(music)
>>> abjad.f(staff)
```

(continues on next page)

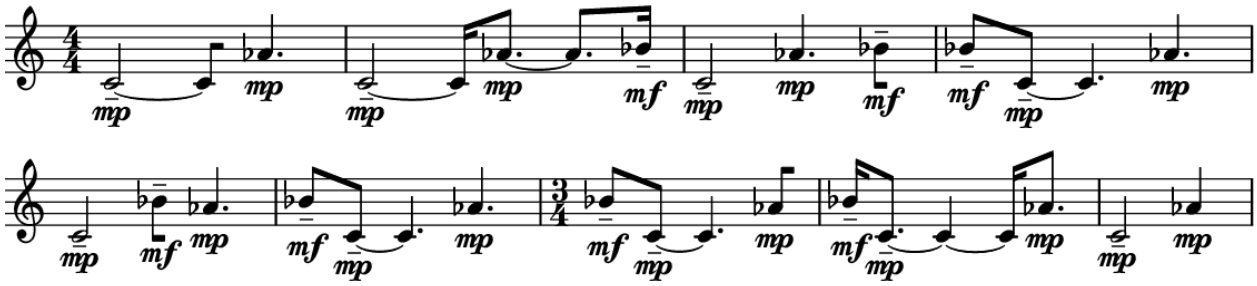
(continued from previous page)

```
\new Staff
{
    \time 4/4
    c'2
    \mp
    - \tenuto
    ~
    c'8
    af'4.
    \mp
    c'2
    \mp
    - \tenuto
    ~
    c'16
    af'8.
    \mp
    ~
    af'8.
    bf'16
    \mf
    - \tenuto
    c'2
    \mp
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
    \mp
    c'2
    \mp
    - \tenuto
    bf'8
    \mf
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
```

(continues on next page)

(continued from previous page)

```
    \mp
    \time 3/4
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'8
    \mp
    bf'16
    \mf
    - \tenuto
    c'8.
    \mp
    - \tenuto
    ~
    c'4
    ~
    c'16
    af'8.
    \mp
    c'2
    \mp
    - \tenuto
    af'4
    \mp
}
```



At this point, let's use `auxjad.remove_repeated_dynamics` to remove all repeated dyanmics. The final result is shown below.

```
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'2
    \mp
    - \tenuto
    ~
    c'8
    af'4.
```

(continues on next page)

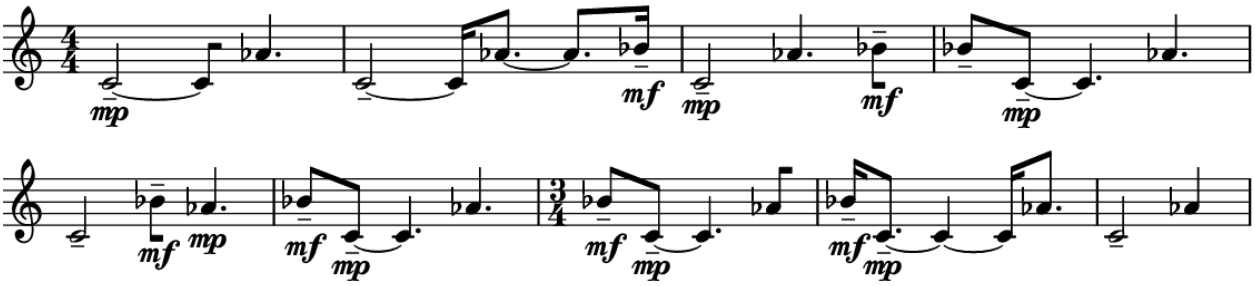
(continued from previous page)

```
    c'2
    - \tenuto
    ~
    c'16
    af'8.
    ~
    af'8.
    bf'16
    \mf
    - \tenuto
    c'2
    \mp
    - \tenuto
    af'4.
    bf'8
    \mf
    - \tenuto
    bf'8
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
    c'2
    - \tenuto
    bf'8
    \mf
    - \tenuto
    af'4.
    \mp
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'4.
    \time 3/4
    bf'8
    \mf
    - \tenuto
    c'8
    \mp
    - \tenuto
    ~
    c'4.
    af'8
    bf'16
    \mf
    - \tenuto
    c'8.
    \mp
    - \tenuto
```

(continues on next page)

(continued from previous page)

```
    ~
    c'4
    ~
    c'16
    af'8.
    c'2
    - \tenuto
    af'4
}
```



1.2 auxjad API

Below is a table with all classes and functions included in `auxjad`. Click on their their names or use the left side panel to navigate to the individual documentaion of each class and function.

auxjad.ArtificialHarmonic(*arguments, ...)	Creates an chord with a tweaked top note head for notating artificial harmonics.
auxjad.CartographySelector(contents, *, ...)	A selector used to store, manipulate, and select objects using a weighted function constructed with a fixed decay rate.
auxjad.close_container(container)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.container_is_full(container)	Returns a `bool` representing whether an input container (of type `abjad.Container` or child class) has its last bar is fully filled in or not.
auxjad.containers_are_equal(container1, ...)	Returns a `bool` representing whether two input containers (of type `abjad.Container` or child class) are identical or not.
auxjad.fill_with_rests(container)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.HarmonicNote(*arguments, multiplier, ...)	Creates a note with tweaked notehead for harmonics.
auxjad.LeafDynMaker(*, increase_monotonic, ...)	This class can be used to create leaves and logical ties from input lists of pitches, durations, dynamics, and articulations.
auxjad.leaves_are_tieable(leaf1, leaf2)	Returns a `bool` representing whether or not two input leaves (of type `abjad.Leaf` or child class) have identical pitch(es) and thus can be tied.

Continued on next page

Table 1 – continued from previous page

auxjad.LoopByList(contents, *, window_size, ...)	This class can be used to output slices of a `list` using the metaphor of a looping window of a variable size.
auxjad.LoopByNotes(contents, *, window_size, ...)	This class can be used to output slices of an `abjad.Container` using the metaphor of a looping window of a variable size.
auxjad.LoopByWindow(contents, *, ...)	This class can be used to output slices of an `abjad.Container` using the metaphor of a looping window of a constant size given by an `abjad.Duration`.
auxjad.remove_repeated_dynamics(container, ...)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.remove_repeated_time_signatures(container)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.respell_container(container, *, ...)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.respell_chord(chord, *, ...)	Mutates an input chord (of type `abjad.Chord` or child class) in place and has no return value.
auxjad.rests_to_multimeasure_rest(container)	Mutates an input container (of type `abjad.Container` or child class) in place and has no return value.
auxjad.Shuffler(contents, *, ...)	Shuffler takes an input `abjad.Container` and shuffles its logical ties.
auxjad.simplified_time_signature_ratio(...)	Returns an `abjad.TimeSignature` with the simplified ratio of an input ratio according to a minimum denominator value.
auxjad.sync_containers(*containers, ...)	Mutates two or more input containers (of type `abjad.Container` or child class) in place and has no return value.
auxjad.TenneySelector(contents, *, weights, ...)	This in an implementation of the Dissonant Counterpoint Algorithm by James Tenney.
auxjad.underfull_duration(container)	Returns the missing `abjad.Duration` of an underfull container (of type `abjad.Container` or child class).

1.2.1 auxjad.ArtificialHarmonic

class auxjad.**ArtificialHarmonic**(**arguments*, *multiplier: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *tag: abjad.system.Tag.Tag = None*, *style: str = 'harmonic'*, *is_parenthesized: bool = False*, *markup: str = None*, *direction: (<class 'str'>, <enum 'VerticalAlignment'>) = 'up'*)

Creates an chord with a tweaked top note head for notating artificial harmonics. This is a child class of `abjad.Chord`.

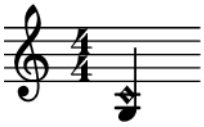
Usage is similar to `abjad.Chord`:

```
>>> harm = auxjad.ArtificialHarmonic("<g c'>4")
>>> harm.style
'harmonic'
>>> abjad.f(harm)
```

(continues on next page)

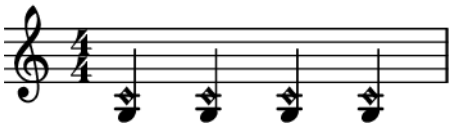
(continued from previous page)

```
<
    g
    \tweak style #'harmonic
    c'
>4
```



And similarly to `abjad.Chord`, pitch and duration can be input in many different ways:

```
>>> harm1 = auxjad.ArtificialHarmonic("<g c'>4")
>>> harm2 = auxjad.ArtificialHarmonic(["g", "c'"], 1/4)
>>> harm3 = auxjad.ArtificialHarmonic([-5, 0], 0.25)
>>> harm4 = auxjad.ArtificialHarmonic([-5, 0], abjad.Duration(1, 4))
>>> staff = abjad.Staff([harm1, harm2, harm3, harm4])
>>> abjad.f(staff)
\new Staff
{
    <
        g
        \tweak style #'harmonic
        c'
    >4
    <
        g
        \tweak style #'harmonic
        c'
    >4
    <
        g
        \tweak style #'harmonic
        c'
    >4
    <
        g
        \tweak style #'harmonic
        c'
    >4
}
```

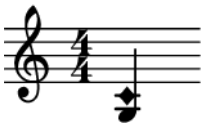


It is important to note that this class can only be initialised with exactly two pitches. Any other number of pitches will raise a `ValueError`:

```
>>> auxjad.ArtificialHarmonic("<g c' d'>4")
ValueError: 'ArtificialHarmonic' requires exactly two 'note_heads' for
initialisation
```

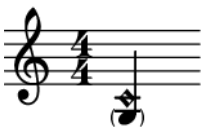
When creating an `ArtificialHarmonic`, use the keyword argument `style` to set a different type of note head for the top note, such as `'harmonic-mixed'`:

```
>>> harm = auxjad.ArtificialHarmonic("<g c'>4",
...                                  style='harmonic-mixed',
...                                  )
>>> harm.style
'harmonic-mixed'
>>> abjad.f(harm)
<
    g
    \tweak style #'harmonic-mixed
    c'
>4
```



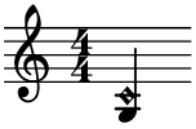
To notate natural harmonics with a parenthesised pitch for the open string at the bottom of the interval, set the keyword `is_parenthesized` to `True`.

```
>>> harm = auxjad.ArtificialHarmonic("<g c'>4",
...                                  is_parenthesized=True,
...                                  )
>>> harm.is_parenthesized
True
>>> abjad.f(harm)
<
    \parenthesize
    \tweak ParenthesesItem.font-size #-4
    g
    \tweak style #'harmonic
    c'
>4
```



Similarly to `abjad.Chord`, `ArtificialHarmonic` can take multipliers:

```
>>> harm = auxjad.ArtificialHarmonic("<g c'>4",
...                                  multiplier=(2, 3),
...                                  )
>>> harm.multiplier
abjad.Multiplier(2, 3)
>>> abjad.f(harm)
<
    g
    \tweak style #'harmonic
    c'
>4 * 2/3
```



All properties of `abjad.Chord` are also available to be read. This class also includes two new properties named `style` and `is_parenthesized`:

```
>>> harm = auxjad.ArtificialHarmonic("<g c'>4")
>>> harm.written_pitches
"g c'"
>>> harm.written_duration
1/4
>>> harm.style
'harmonic'
>>> harm.is_parenthesized
False
```

All these properties can be set to different values after initialisation:

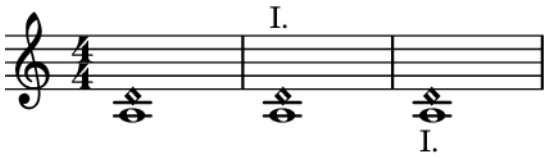
```
>>> harm.written_pitches = [-5, 2]
>>> harm.written_duration = abjad.Duration(1, 8)
>>> harm.style = 'harmonic-mixed'
>>> harm.is_parenthesized = True
>>> harm.written_pitches
"g d'"
>>> harm.written_duration
1/8
>>> harm.style
'harmonic-mixed'
>>> harm.is_parenthesized
True
```

The methods `sounding_pitch()` and `sounding_note()` return the sounding pitch and sounding note, respectively. Their types are `abjad.Pitch` and `abjad.Note`, respectively.

```
>>> harmonics = [ArtificialHarmonic("<g b>4"),
...              ArtificialHarmonic("<g c'>4"),
...              ArtificialHarmonic("<g d'>4"),
...              ArtificialHarmonic("<g e'>4"),
...              ArtificialHarmonic("<g g'>4"),
...              ]
>>> for harmonic in harmonics:
...     print(harmonic.sounding_pitch())
b''
g''
d''
b''
g'
>>> for harmonic in harmonics:
...     print(harmonic.sounding_note())
b''4
g''4
d''4
b''4
g'4
```

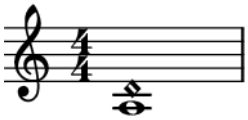
To add a markup expression to the harmonic note, use the markup:

```
>>> harm1 = auxjad.ArtificialHarmonic("<a d'>1")
>>> harm2 = auxjad.ArtificialHarmonic("<a d'>1",
...                                   markup='I.',
...                                   )
>>> harm3 = auxjad.ArtificialHarmonic("<a d'>1",
...                                   markup='I.',
...                                   direction=abjad.Down)
>>> staff = abjad.Staff([harm1, harm2, harm3])
>>> abjad.f(staff)
\new Staff
{
    <
        a
        \tweak style #'harmonic
        d'
    >1
    <
        a
        \tweak style #'harmonic
        d'
    >1
    ^ \markup { I. }
    <
        a
        \tweak style #'harmonic
        d'
    >1
    _ \markup { I. }
}
```



Setting `markup` to `None` will remove the markup from the note.

```
>>> harm = auxjad.ArtificialHarmonic("<a d'>1",
...                                  markup='I.',
...                                  )
>>> harm.markup = None
>>> abjad.f(harm)
<
    a
    \tweak style #'harmonic
    d'
>1
```

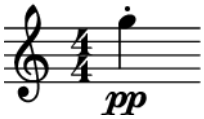


Warning: If another markup is attached to the harmonic note, trying to set the `markup` to `None` will raise an Exception:

```
>>> harm = auxjad.ArtificialHarmonic("<a d'>1")
>>> abjad.attach(abjad.Markup('test'), harm)
>>> harm.markup = 'I.'
>>> harm.markup = None
Exception: multiple indicators attached to client.
```

The note created by `sounding_note()` inherits all indicators from the `ArtificialHarmonic`.

```
>>> harm = auxjad.ArtificialHarmonic(r"<g c'>4-.\pp")
>>> abjad.f(harm.sounding_note())
g''4
\pp
- \staccato
```



Warning: Both `sounding_pitch()` and `sounding_note()` methods raise a ValueError exception when it cannot calculate the sounding pitch for the given interval.

```
>>> ArtificialHarmonic("<g ef'>4").sounding_pitch()
ValueError: cannot calculate sounding pitch for given interval
>>> ArtificialHarmonic("<g ef'>4").sounding_note()
ValueError: cannot calculate sounding pitch for given interval
```

__init__ (**arguments*, *multiplier: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *tag: abjad.system.Tag.Tag = None*, *style: str = 'harmonic'*, *is_parenthesized: bool = False*, *markup: str = None*, *direction: (<class 'str'>, <enum 'VerticalAlignment'>) = 'up'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*arguments, multiplier, Tuple[int, ...)	Initialize self.
`sounding_note`()	Returns the sounding note of the harmonic as an `abjad.Note`.
`sounding_pitch`()	Returns the sounding pitch of the harmonic as an `abjad.Pitch`.

Attributes

`direction`	The direction of the harmonic note head.
`is_parenthesized`	Whether the bottom note head is parenthesised or not.

Continued on next page

Table 3 – continued from previous page

markup	The markup of the harmonic note head.
multiplier	Gets multiplier.
note_heads	Gets note-heads in chord.
style	The style of the upper note head.
tag	Gets component tag.
written_duration	Gets and sets written duration of chord.
written_pitches	The written pitches of the two note heads.

1.2.2 auxjad.CartographySelector

class auxjad.**CartographySelector**(*contents: list*, *, *decay_rate: float = 0.75*)

A selector used to store, manipulate, and select objects using a weighted function constructed with a fixed decay rate. The decay rate represents the ratio of probabilities of any index given the probability of the preceeding one. For instance, if the decay rate is set to `0.75` (which is its default value), the probability of the element in index 1 of the input list being selected is 0.75 the probability of the element in index 0, and the probability of the element in index 2 is 0.5625 (0.75^2) the probability of the element in index 0. This is the selector used in my *Cartography* series of compositions.

The selector should be initialised with a list of objects. The contents of the list can be absolutely anything.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
```

The default decay rate is 0.75; that is, the weight of any given elements is the weight of the previous one multiplied by 0.75. The weights are associated with the index position, not the elements themselves.

```
>>> selector.weights
[1.0, 0.75, 0.5625, 0.421875, 0.31640625]
```

Applying the `len()` function to the selector will give the length of the input list.

```
>>> len(selector)
5
```

Calling the selector will output one of its elements, selected according to its weight function.

```
>>> selector()
2
```

Alternatively, use the `next()` function or `__next__()` method to get the next result.

```
>>> selector.__next__()
1
>>> next(selector)
0
```

By default, only the weight function (defined by the decay rate) is taken into consideration when selecting an element. This means that repeated elements can appear, as shown below.

```
>>> result = ''
>>> for _ in range(30):
...     result += str(selector())
>>> result
203001402200011111101400310140
```

Calling the selector with the optional keyword argument `no_repeat` set to `True` will forbid immediate repetitions among consecutive calls.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> result = ''
>>> for _ in range(30):
...     result += str(selector(no_repeat=True))
>>> result
210421021020304024230120241202
```

The keyword argument `decay_rate` can be used to set a different decay rate when creating a selector.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4],
...                                       decay_rate=0.5,
...                                       )
>>> selector.weights
[1.0, 0.5, 0.25, 0.125, 0.0625]
```

The decay rate can also be set after the creation of a selector using, the property `decay_rate`.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.decay_rate = 0.2
>>> selector.weights
[1.0, 0.2, 0.04000000000000001, 0.008000000000000002,
0.0016000000000000003]
>>> result = ''
>>> for _ in range(30):
...     result += str(selector())
>>> result
'000001002100000201001030000100'
```

Appending is a type of content transformation. It discards the first element of the selector's `contents`, shifts all others leftwards, and then appends the new element to the last index.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.append(5)
>>> selector.contents
[1, 2, 3, 4, 5]
>>> selector.append(42)
>>> selector.contents
[2, 3, 4, 5, 42]
```

The method `append_keeping_n()` is similar to `append()`, but it keeps the first `n` elements of `contents` untouched. It thus discards the n+1-th element, shifts all the next elements one position lefwards, and finally appends the new element at the last index.

```
>>> selector = auxjad.CartographySelector([10, 7, 14, 31, 98])
>>> selector.contents
[10, 7, 14, 31, 98]
>>> selector.append_keeping_n(100, 2)
>>> selector.contents
[10, 7, 31, 98, 100]
```

Prepending is another type of content transformation. It discards the last element of the `contents`, shifts all others rightwards, and then prepends the new element to the first index.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.prepend(-1)
>>> selector.contents
[-1, 0, 1, 2, 3]
>>> selector.prepend(71)
>>> selector.contents
[71, -1, 0, 1, 2]
```

Rotation is another type of content transformation. It rotates all elements rightwards, moving the last element to the first index. If the optional keyword argument `anticlockwise` is set to `True`, the rotation will be in the opposite direction.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.rotate()
>>> selector.contents
[1, 2, 3, 4, 0]
>>> selector.rotate(anticlockwise=True)
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.rotate(anticlockwise=True)
>>> selector.contents
[1, 2, 3, 4, 0]
```

The mirror transformation swaps the element of the input index with its complementary element. Complementary elements are defined as the pair of elements which share the same distance from the centre of the `contents` (in terms of number of indeces), and are located at either side.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.mirror(0)
>>> selector.contents
[4, 1, 2, 3, 0]
>>> selector.mirror(0)
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.mirror(3)
>>> selector.contents
[0, 3, 2, 1, 4]
>>> selector.mirror(2)
>>> selector.contents
[0, 3, 2, 1, 4]
```

To mirror a random pair of complementary elements, use the `mirror_random()` method. In case of a selector with an odd number of elements, this method will never pick an element at the pivot point since the operation would not change the contents.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.mirror_random()
>>> selector.contents
[4, 1, 2, 3, 0]
```

(continues on next page)

(continued from previous page)

```
>>> selector.mirror_random()
>>> selector.contents
[4, 3, 2, 1, 0]
>>> selector.mirror_random()
>>> selector.contents
[4, 1, 2, 3, 0]
```

The method `randomise()` will randomise the position of the elements of a selector's `contents`.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4])
>>> selector.contents
[0, 1, 2, 3, 4]
>>> selector.randomise()
>>> selector.contents
[1, 4, 3, 0, 2]
```

The contents of a selector can also be altered after it has been initialised using the `contents` property. The length of the contents can change as well.

```
>>> selector = auxjad.CartographySelector([0, 1, 2, 3, 4],
...                                       decay_rate=0.5,
...                                       )
>>> len(selector)
5
>>> selector.weights
[1.0, 0.5, 0.25, 0.125, 0.0625]
>>> selector.contents = [10, 7, 14, 31, 98, 47, 32]
>>> selector.contents
[10, 7, 14, 31, 98, 47, 32]
>>> len(selector)
7
>>> selector.weights
[1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625]
```

Use the read-only properties `previous_result` and `previous_index` to output the previous result and its index.

```
>>> selector = auxjad.CartographySelector([10, 7, 14, 31, 98])
>>> selector()
14
>>> previous_index = selector.previous_index
>>> previous_index
2
>>> selector.previous_result
14
```

This class allows indecing and slicing just like regular lists. This can be used to both access and alter elements.

```
>>> selector = auxjad.CartographySelector([10, 7, 14, 31, 98])
>>> selector[1]
7
>>> selector[1:4]
[7, 14, 31]
>>> selector[:]
[10, 7, 14, 31, 98]
>>> selector()
```

(continues on next page)

(continued from previous page)

```
31
>>> previous_index = selector.previous_index
>>> previous_index
3
>>> selector[previous_index]
31
>>> selector.contents
[10, 7, 14, 31, 98]
>>> selector[2] = 100
>>> selector.contents
[10, 7, 100, 31, 98]
```

__init__ (*contents: list*, ***, *decay_rate: float = 0.75*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, decay_rate)	Initialize self.
append(new_element)	A type of content transformation, it discards the first element of the contents, shifts all others leftwards, and then appends the new element to the last index.
append_keeping_n(new_element, n)	A type of content transformation similar to append(), it keeps the first n elements of contents untouched, it then discards the n+1-th element, shifts all the next elements one position lefwards, and finally appends the new element at the last index.
mirror(index)	A type of content transformation which swaps the element of the input index with its complementary element.
mirror_random()	A type of content transformation which swaps the element of a random index with its complementary element.
prepend(new_element)	A type of content transformation, it discards the last element of the contents, shifts all others rightwards, and then prepends the new element to the first index.
randomise()	Randomises the position of the elements of contents.
rotate([anticlockwise])	A type of content transformation, it rotates all elements rightwards, moving the last element to the first index.

Attributes

contents	The list from which the selector picks elements.
decay_rate	The decay rate represents the ratio of probabilities of any index given the probability of the preceeding one.

Continued on next page

Table 5 – continued from previous page

previous_index	Read-only property, returns the index of the previously output element.
previous_result	Read-only property, returns the previously output element.

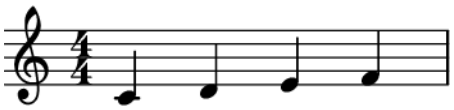
1.2.3 auxjad.close_container

auxjad.**close_container**(*container: abjad.core.Container.Container*)

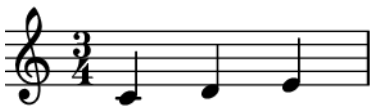
Mutates an input container (of type abjad.Container or child class) in place and has no return value. This function changes the time signature of the last bar of an underfull in order to make it full.

Returns the missing duration of the last bar of any container or child class. If no time signature is encountered, it uses LilyPond's convention and considers the container as in 4/4.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.close_container(container1)
>>> auxjad.close_container(container2)
>>> auxjad.close_container(container3)
>>> auxjad.close_container(container4)
>>> abjad.f(container1)
{
    c'4
    d'4
    e'4
    f'4
}
```



```
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    c'4
    d'4
    e'4
}
```

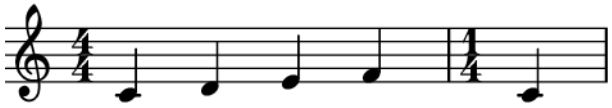


```
>>> abjad.f(container3)
{
    c'4
    d'4
    e'4
    f'4
```

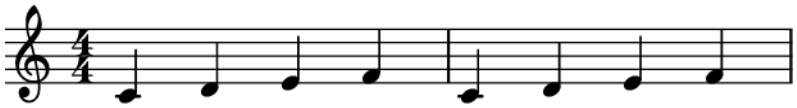
(continues on next page)

(continued from previous page)

```
    %%% \time 1/4 %%%
    c'4
}
```

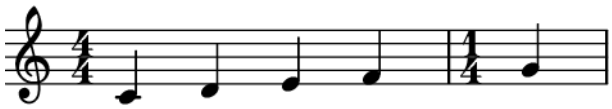


```
>>> abjad.f(container4)
{
    c'4
    d'4
    e'4
    f'4
    c'4
    d'4
    e'4
    f'4
}
```



Handles any time signatures as well as changes of time signature.

```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4 g'")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 2/4 c'4")
>>> container3 = abjad.Container(r"\time 5/4 g1 ~ g4 \time 4/4 af'2")
>>> auxjad.close_container(container1)
>>> auxjad.close_container(container2)
>>> auxjad.close_container(container3)
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    c'4
    d'4
    e'4
    f'4
    %%% \time 1/4 %%%
    g'4
}
```

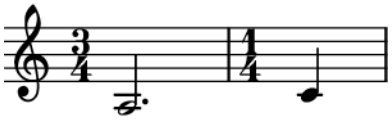


```
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    a2.
```

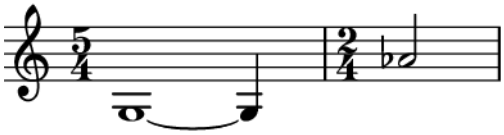
(continues on next page)

(continued from previous page)

```
    %%% \time 1/4 %%%
    c'4
}
```



```
>>> abjad.f(container3)
{
    %%% \time 5/4 %%%
    g1
    ~
    g4
    %%% \time 2/4 %%%
    af'2
}
```

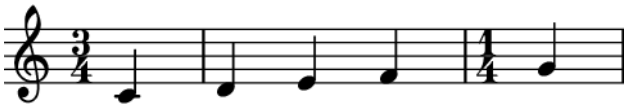


Note: Notice that the time signatures in the output are commented out with `%%%`. This is because Abjad only applies time signatures to containers that belong to a `abjad.Staff`. The present function works with either `abjad.Container` and `abjad.Staff`.

```
>>> container = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4 g'")
>>> auxjad.close_container(container)
>>> abjad.f(container)
{
    %%% \time 4/4 %%%
    c'4
    d'4
    e'4
    f'4
    %%% \time 1/4 %%%
    g'4
}
>>> staff = abjad.Staff([container])
>>> abjad.f(container)
{
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    \time 1/4
    g'4
}
```

Correctly handles partial time signatures.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4 g'4")
>>> time_signature = abjad.TimeSignature((3, 4), partial=(1, 4))
>>> abjad.attach(time_signature, container[0])
>>> auxjad.close_container(container)
>>> abjad.f(container)
{
    %%% \partial 4 %%%
    %%% \time 3/4 %%%
    c'4
    d'4
    e'4
    f'4
    %%% \time 1/4 %%%
    g'4
}
```



Warning: If a container is malformed, i.e. it has an underfilled bar before a time signature change, the function raises a `ValueError` exception.

```
>>> container = abjad.Container(r"\time 5/4 g''1 \time 4/4 f'4")
>>> auxjad.close_container(container)
ValueError: 'container' is malformed, with an underfull bar preceeding
a time signature change
```

1.2.4 auxjad.container_is_full

auxjad.**container_is_full**(*container: abjad.core.Container.Container*) → bool

Returns a `bool` representing whether an input container (of type `abjad.Container` or child class) has its last bar is fully filled in or not.

Returns `True` if the last bar of any container (or child class) is full, otherwise returns `False`. If no time signature is encountered, it uses LilyPond's convention and considers the container as in 4/4.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.container_is_full(container1)
True
>>> auxjad.container_is_full(container2)
False
>>> auxjad.container_is_full(container3)
False
>>> auxjad.container_is_full(container4)
True
```

Handles any time signatures as well as changes of time signature.

```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 2/4 r2")
>>> container3 = abjad.Container(r"\time 5/4 g1 ~ g4 \time 4/4 af'2")
>>> container4 = abjad.Container(r"\time 6/8 c'2 ~ c'8")
>>> auxjad.container_is_full(container1)
True
>>> auxjad.container_is_full(container2)
True
>>> auxjad.container_is_full(container3)
False
>>> auxjad.container_is_full(container4)
False
```

Correctly handles partial time signatures.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> time_signature = abjad.TimeSignature((3, 4), partial=(1, 4))
>>> abjad.attach(time_signature, container[0])
>>> auxjad.container_is_full(container)
True
```

It also handles multi-measure rests.

```
>>> container1 = abjad.Container(r"R1")
>>> container2 = abjad.Container(r"\time 3/4 R1*3/4 \time 2/4 r2")
>>> container3 = abjad.Container(r"\time 5/4 R1*5/4 \time 4/4 g''4")
>>> container4 = abjad.Container(r"\time 6/8 R1*1/2")
>>> auxjad.container_is_full(container1)
True
>>> auxjad.container_is_full(container2)
True
>>> auxjad.container_is_full(container3)
False
>>> auxjad.container_is_full(container4)
False
```

Warning: If a container is malformed, i.e. it has an underfilled bar before a time signature change, the function raises a `ValueError` exception.

```
>>> container = abjad.Container(r"\time 5/4 g''1 \time 4/4 f'1")
>>> auxjad.container_is_full(container)
ValueError: 'container' is malformed, with an underfull bar preceeding
a time signature change
```

1.2.5 auxjad.containers_are_equal

auxjad.**containers_are_equal**(*container1: abjad.core.Container.Container*, *container2: abjad.core.Container.Container*, **, include_indicators: bool = False*) → bool

Returns a `bool` representing whether two input containers (of type `abjad.Container` or child class) are identical or not.

When the pitches and effective durations of all leaves in both containers are identical, this function returns `True`:

```
>>> container1 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
True
```

Even if all leaves of both containers are identical in relation to both pitches and written durations, the function considers the effective durations. This means that situations like the one below do not yield a false positive:

```
>>> container1 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"\times 3/2 {c'4 d'4 e'4} "
...                          "f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
False
```

By default, this function ignores indicators, so the containers in the example below are understood to be identical:

```
>>> container1 = abjad.Staff(r"c'4\pp d'4 e'4-. f'4 <g' a'>2-> r2")
>>> container2 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
True
```

Setting the argument `include_indicators` to `True` forces the function to include indicators in its comparison. In that case, the containers in the example above are not considered identical any longer:

```
>>> container1 = abjad.Staff(r"c'4\pp d'4 e'4-. f'4 <g' a'>2-> r2")
>>> container2 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1,
...                             container2,
...                             include_indicators=True,
...                             )
True
```

This function also handles grace notes:

```
>>> container1 = abjad.Staff(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Staff(r"c'4 \grace{d'4} d'4 e'4 f'4")
>>> auxjad.containers_are_equal(container1, container2)
False
```

```
>>> container1 = abjad.Staff(r"c'4 d'4 e'4 f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"c'4 \grace{c''4} d'4 e'4 "
...                          "f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
False
```

```
>>> container1 = abjad.Staff(r"c'4 \grace{c''4} d'4 e'4 "
...                          "f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"c'4 \grace{c''8} d'4 e'4 "
...                          "f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
False
```

```
>>> container1 = abjad.Staff(r"c'4 \grace{c''16} d'4 e'4 "
...                          "f'4 <g' a'>2 r2")
>>> container2 = abjad.Staff(r"c'4 \grace{c''16} d'4 e'4 "
```

(continues on next page)

(continued from previous page)

```
...                             "f'4 <g' a'>2 r2")
>>> auxjad.containers_are_equal(container1, container2)
True
```

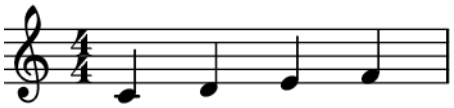
1.2.6 auxjad.fill_with_rests

auxjad.**fill_with_rests**(*container: abjad.core.Container.Container*)

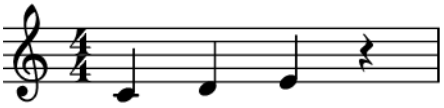
Mutates an input container (of type abjad.Container or child class) in place and has no return value. This function fills a container with rests in order to make it full.

Returns the missing duration of the last bar of any container or child class. If no time signature is encountered, it uses LilyPond's convention and considers the container as in 4/4.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.fill_with_rests(container1)
>>> auxjad.fill_with_rests(container2)
>>> auxjad.fill_with_rests(container3)
>>> auxjad.fill_with_rests(container4)
>>> abjad.f(container1)
{
    c'4
    d'4
    e'4
    f'4
}
```



```
>>> abjad.f(container2)
{
    c'4
    d'4
    e'4
    r4
}
```

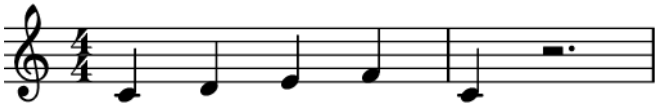


```
>>> abjad.f(container3)
{
    c'4
    d'4
    e'4
    f'4
```

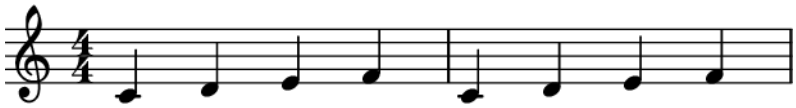
(continues on next page)

(continued from previous page)

```
    c'4
    r2.
}
```

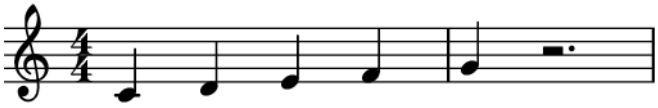


```
>>> abjad.f(container4)
{
    c'4
    d'4
    e'4
    f'4
    c'4
    d'4
    e'4
    f'4
}
```



Handles any time signatures as well as changes of time signature.

```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4 g'")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 2/4 c'4")
>>> container3 = abjad.Container(r"\time 5/4 g1 ~ g4 \time 4/4 af'2")
>>> auxjad.fill_with_rests(container1)
>>> auxjad.fill_with_rests(container2)
>>> auxjad.fill_with_rests(container3)
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    c'4
    d'4
    e'4
    f'4
    g'4
    r2.
}
```

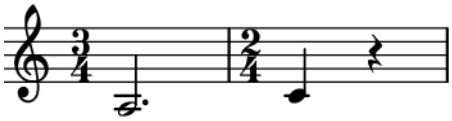


```
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    a2.
```

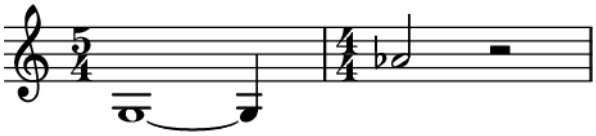
(continues on next page)

(continued from previous page)

```
    %%% \time 2/4 %%%
    c'4
    r4
}
```



```
>>> abjad.f(container3)
{
    %%% \time 5/4 %%%
    g1
    ~
    g4
    %%% \time 4/4 %%%
    af'2
    r2
}
```

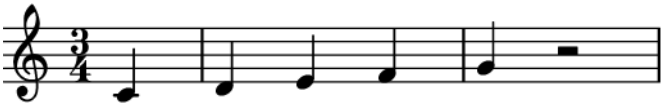


Note: Notice that the time signatures in the output are commented out with `%%%`. This is because Abjad only applies time signatures to containers that belong to a `abjad.Staff`. The present function works with either `abjad.Container` and `abjad.Staff`.

```
>>> container = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4 g'")
>>> auxjad.fill_with_rests(container)
>>> abjad.f(container)
{
    %%% \time 4/4 %%%
    c'4
    d'4
    e'4
    f'4
    g'4
    r2.
}
>>> staff = abjad.Staff([container])
>>> abjad.f(container)
{
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    g'4
    r2.
}
```

Correctly handles partial time signatures.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4 g'4")
>>> time_signature = abjad.TimeSignature((3, 4), partial=(1, 4))
>>> abjad.attach(time_signature, container[0])
>>> auxjad.fill_with_rests(container)
>>> abjad.f(container)
{
    %%% \partial 4 %%%
    %%% \time 3/4 %%%
    c'4
    d'4
    e'4
    f'4
    g'4
    r2
}
```



Warning: If a container is malformed, i.e. it has an underfilled bar before a time signature change, the function raises a `ValueError` exception.

```
>>> container = abjad.Container(r"\time 5/4 g''1 \time 4/4 f'4")
>>> auxjad.fill_with_rests(container)
ValueError: 'container' is malformed, with an underfull bar preceeding
a time signature change
```

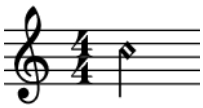
1.2.7 auxjad.HarmonicNote

class auxjad.**HarmonicNote**(**arguments*, *multiplier: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *tag: abjad.system.Tag.Tag = None*, *style: str = 'harmonic'*, *markup: str = None*, *direction: (<class 'str'>, <enum 'VerticalAlignment'>) = 'up'*)

Creates a note with tweaked notehead for harmonics. This is a child class of `abjad.Note`.

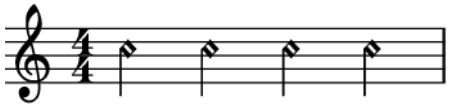
Usage is similar to `abjad.Note`:

```
>>> harm = auxjad.HarmonicNote("c''4")
>>> harm.style
'harmonic'
>>> abjad.f(harm)
\tweak style #'harmonic
c''4
```



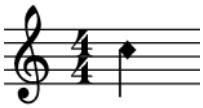
And similarly to `abjad.Note`, pitch and duration can be input in many different ways:

```
>>> harm1 = auxjad.HarmonicNote("c''4")
>>> harm2 = auxjad.HarmonicNote("c''", 1/4)
>>> harm3 = auxjad.HarmonicNote(12, 0.25)
>>> harm4 = auxjad.HarmonicNote(12, abjad.Duration(1, 4))
>>> staff = abjad.Staff([harm1, harm2, harm3, harm4])
>>> abjad.f(staff)
\new Staff
{
    \tweak style #'harmonic
    c''4
    \tweak style #'harmonic
    c''4
    \tweak style #'harmonic
    c''4
    \tweak style #'harmonic
    c''4
}
```



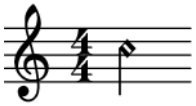
When creating an `HarmonicNote`, use the keyword argument `style` to set a different type of note head, such as `'harmonic-mixed'`:

```
>>> harm = auxjad.HarmonicNote("c''4",
...                            style='harmonic-mixed',
...                            )
>>> harm.style
'harmonic-mixed'
>>> abjad.f(harm)
\tweak style #'harmonic-mixed
c''4
```



Similarly to `abjad.Note`, `HarmonicNote` can take multipliers:

```
>>> harm = auxjad.HarmonicNote("c''4",
...                            multiplier=(2, 3),
...                            )
>>> harm.multiplier
abjad.Multiplier(2, 3)
>>> abjad.f(harm)
\tweak style #'harmonic
c''4 * 2/3
```



All properties of `abjad.Note` are also available to be read. This class also includes a new property named `style`:

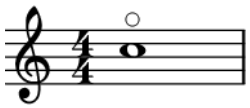
```
>>> harm = auxjad.HarmonicNote("c''4")
>>> harm.written_pitch
"c''"
>>> harm.written_duration
1/4
>>> harm.style
'harmonic'
```

All these properties can be set to different values after initialisation:

```
>>> harm.written_pitch = 18
>>> harm.written_duration = abjad.Duration(1, 8)
>>> harm.style = 'harmonic-mixed'
>>> harm.written_pitch
"fs''"
>>> harm.written_duration
1/8
>>> harm.style
'harmonic-mixed'
```

To create a harmonic note with a regular note head and with a flageolet circle above it, use the style `'flageolet'`:

```
>>> harm = auxjad.HarmonicNote("c''1",
...                            style='flageolet',
...                            )
>>> harm.style
'flageolet'
>>> abjad.f(harm)
c''1
\flageolet
```



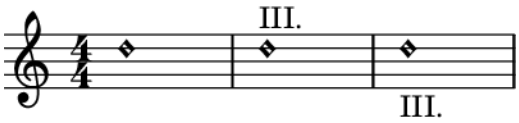
To add a markup expression to the harmonic note, use the markup:

```
>>> harm1 = auxjad.HarmonicNote("d''1")
>>> harm2 = auxjad.HarmonicNote("d''1",
...                             markup='III.',
...                             )
>>> harm3 = auxjad.HarmonicNote("d''1",
...                             markup='III.',
...                             direction=abjad.Down)
>>> staff = abjad.Staff([harm1, harm2, harm3])
>>> abjad.f(staff)
\new Staff
{
    \tweak style #'harmonic
    d''1
    \tweak style #'harmonic
    d''1
```

(continues on next page)

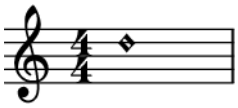
(continued from previous page)

```
    ^ \markup { III. }
    \tweak style #'harmonic
    d''1
    _ \markup { III. }
}
```



Setting `markup` to `None` will remove the markup from the note.

```
>>> harm = auxjad.HarmonicNote("d''1",
...                            markup='III.',
...                            )
>>> harm.markup = None
>>> abjad.f(harm)
\tweak style #'harmonic
d''1
```



Warning: If another markup is attached to the harmonic note, trying to set the `markup` to `None` will raise an Exception:

```
>>> harm = auxjad.HarmonicNote("d''1")
>>> abjad.attach(abjad.Markup('test'), harm)
>>> harm.markup = 'III.'
>>> harm.markup = None
Exception: multiple indicators attached to client.
```

__init__(**arguments*, *multiplier: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *tag: abjad.system.Tag.Tag = None*, *style: str = 'harmonic'*, *markup: str = None*, *direction: (<class 'str'>, <enum 'VerticalAlignment'>) = 'up'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*arguments, multiplier, Tuple[int, ...)	Initialize self.
from_pitch_and_duration(pitch, duration)	Makes note from `pitch` and `duration`.

Attributes

direction	The direction of the harmonic note head.

Continued on next page

Table 7 – continued from previous page

markup	The markup of the harmonic note head.
multiplier	Gets multiplier.
note_head	Gets and sets note-head.
style	The style of the harmonic note head.
tag	Gets component tag.
written_duration	Gets and sets written duration.
written_pitch	Gets and sets written pitch.

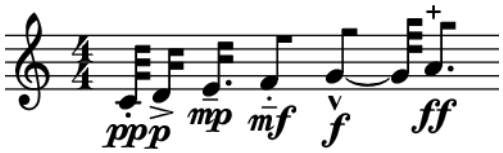
1.2.8 auxjad.LeafDynMaker

class auxjad.**LeafDynMaker**(*, *increase_monotonic: bool = None*, *forbidden_note_duration: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *forbidden_rest_duration: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *skips_instead_of_rests: bool = None*, *tag: abjad.system.Tag.Tag = None*, *use_multimeasure_rests: bool = None*)

This class can be used to create leaves and logical ties from input lists of pitches, durations, dynamics, and articulations. It is an extension of `abjad.LeafMaker` which can take optional lists of dynamics and articulations.

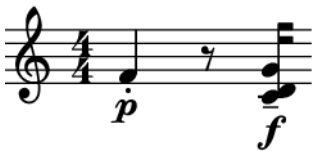
Usage is similar to `abjad.LeafMaker`:

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = [(1, 32), (2, 32), (3, 32), (4, 32), (5, 32), (6, 32)]
>>> dynamics = ['pp', 'p', 'mp', 'mf', 'f', 'ff']
>>> articulations = ['.', '>', '-', '_', '^', '+']
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'32
    \pp
    -\staccato
    d'16
    \p
    -\accent
    e'16.
    \mp
    -\tenuto
    f'8
    \mf
    -\portato
    g'8
    \f
    -\marcato
    ~
    g'32
    a'8.
    \ff
    -\stopped
}
```



Tuple elements in `pitches` result in chords. None-valued elements in `pitches` result in rests:

```
>>> pitches = [5, None, (0, 2, 7)]
>>> durations = [(1, 4), (1, 8), (1, 16)]
>>> dynamics = ['p', None, 'f']
>>> articulations = ['staccato', None, 'tenuto']
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    f'4
    \p
    -\staccato
    r8
    <c' d' g'>16
    \f
    -\tenuto
}
```



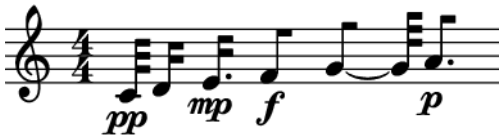
Can omit repeated dynamics with the keyword argument `omit_repeated_dynamics`:

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = [(1, 32), (2, 32), (3, 32), (4, 32), (5, 32), (6, 32)]
>>> dynamics = ['pp', 'pp', 'mp', 'f', 'f', 'p']
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches,
...                        durations,
...                        dynamics,
...                        omit_repeated_dynamics=True,
...                        )
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'32
    \pp
    d'16
    e'16.
    \mp
    f'8
    \f
    g'8
```

(continues on next page)

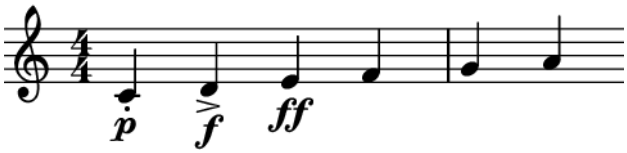
(continued from previous page)

```
    ~
    g'32
    a'8.
    \p
}
```



The lengths `dynamics` and `articulations` can be shorter than the lengths of `pitches` and `durations` (whatever is the greatest):

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = (1, 4)
>>> dynamics = ['p', 'f', 'ff']
>>> articulations = ['.', '>']
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \p
    -\staccato
    d'4
    \f
    -\accent
    e'4
    \ff
    f'4
    g'4
    a'4
}
```



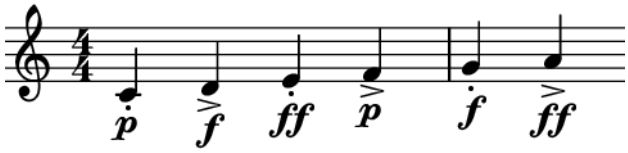
If the lengths of either `dynamics` and `articulations` are shorter than the lengths of `pitches` and `durations` (whatever is the greatest), use the optional keyword arguments `cyclic_dynamics` and `cyclic_articulations` to apply those parameters cyclically:

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = (1, 4)
>>> dynamics = ['p', 'f', 'ff']
>>> articulations = ['.', '>']
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches,
```

(continues on next page)

(continued from previous page)

```
...                         durations,
...                         dynamics,
...                         articulations,
...                         cyclic_dynamics=True,
...                         cyclic_articulations=True,
...                         )
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \p
    - \staccato
    d'4
    \f
    - \accent
    e'4
    \ff
    - \staccato
    f'4
    \p
    - \accent
    g'4
    \f
    - \staccato
    a'4
    \ff
    - \accent
}
```



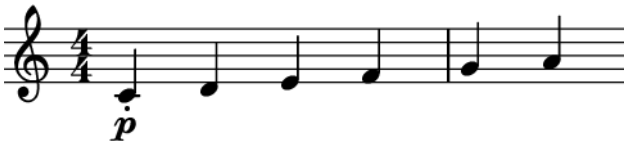
If the length of `articulations` or `dynamics` is 1, they will be applied only to the first element.

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = (1, 4)
>>> dynamics = 'p'
>>> articulations = '.'
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \p
    -\staccato
    d'4
    e'4
    f'4
    g'4
```

(continues on next page)

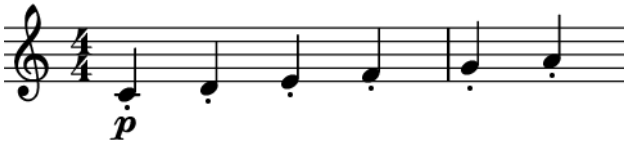
(continued from previous page)

```
    a'4
}
```



To apply them to all elements, use the `cyclic_dynamics` and `cyclic_articulations` optioanl keywords.

```
>>> pitches = [0, 2, 4, 5, 7, 9]
>>> durations = (1, 4)
>>> dynamics = 'p'
>>> articulations = '.'
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches,
...                        durations,
...                        dynamics,
...                        articulations,
...                        cyclic_articulations=True,
...                        )
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \p
    -\staccato
    d'4
    -\staccato
    e'4
    -\staccato
    f'4
    -\staccato
    g'4
    -\staccato
    a'4
    -\staccato
}
```



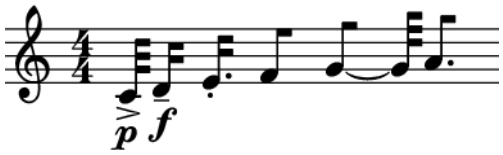
Similarly to Abjad's native classes, it accepts many types of elements in its input lists:

```
>>> pitches = [0,
...            "d'",
...            'E4',
...            abjad.NumberedPitch(5),
...            abjad.NamedPitch("g'"),
```

(continues on next page)

(continued from previous page)

```
...             abjad.NamedPitch("A4"),
...             ]
>>> durations = [(1, 32),
...              "2/32",
...              abjad.Duration("3/32"),
...              abjad.Duration(0.125),
...              abjad.Duration(5, 32),
...              abjad.Duration(6/32),
...              ]
>>> dynamics = ['p',
...             abjad.Dynamic('f'),
...             ]
>>> articulations = ['>',
...                  abjad.Articulation('-'),
...                  abjad.Staccato(),
...                  ]
>>> leaf_dyn_maker = auxjad.LeafDynMaker()
>>> notes = leaf_dyn_maker(pitches, durations, dynamics, articulations)
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'32
    \p
    - \accent
    d'16
    \f
    - \tenuto
    e'16.
    \staccato
    f'8
    g'8
    ~
    g'32
    a'8.
}
```



__init__(*, *increase_monotonic: bool = None*, *forbidden_note_duration: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *forbidden_rest_duration: Union[abjad.utilities.Duration.Duration, Tuple[int, int]] = None*, *skips_instead_of_rests: bool = None*, *tag: abjad.system.Tag.Tag = None*, *use_multimeasure_rests: bool = None*) → None

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(*, increase_monotonic, ...)	Initialize self.

Attributes

forbidden_note_duration	Gets forbidden written duration.
forbidden_rest_duration	Gets forbidden written duration.
increase_monotonic	Is true when durations increase monotonically.
skips_instead_of_rests	Is true when skips appear in place of rests.
tag	Gets tag.
use_multimeasure_rests	Is true when rests are multimeasure.

1.2.9 auxjad.leaves_are_tieable

auxjad.**leaves_are_tieable**(*leaf1: abjad.core.Leaf.Leaf*, *leaf2: abjad.core.Leaf.Leaf*) → bool

Returns a `bool` representing whether or not two input leaves (of type `abjad.Leaf` or child class) have identical pitch(es) and thus can be tied.

When the pitches in both leaves are identical, this function returns `True`:

```
>>> Leaf1 = abjad.Note(r"c'4")
>>> Leaf2 = abjad.Note(r"c'4")
>>> auxjad.leaves_are_tieable(Leaf1, Leaf2)
True
```

Durations do not affect the comparison.

```
>>> Leaf1 = abjad.Note(r"c'2.")
>>> Leaf2 = abjad.Note(r"c'16")
>>> Leaf3 = abjad.Note(r"f'''16")
>>> auxjad.leaves_are_tieable(Leaf1, Leaf2)
True
>>> auxjad.leaves_are_tieable(Leaf1, Leaf3)
False
>>> auxjad.leaves_are_tieable(Leaf2, Leaf3)
False
```

Handles chords as well as pitches.

```
>>> chord1 = abjad.Chord(r"<c' e' g'>4")
>>> chord2 = abjad.Chord(r"<c' e' g'>16")
>>> chord3 = abjad.Chord(r"<f''' fs'''>16")
>>> auxjad.leaves_are_tieable(chord1, chord2)
True
>>> auxjad.leaves_are_tieable(chord1, chord3)
False
>>> auxjad.leaves_are_tieable(chord2, chord3)
False
```

Leaves can also be part of containers.

```
>>> container = abjad.Container(r"r4 <c' e'>4 <c' e'>2")
>>> auxjad.leaves_are_tieable(container[1], container[2])
True
```

If rests are input, the return value is `False`.

```
>>> container = abjad.Container(r"r4 g'4 r2")
>>> auxjad.leaves_are_tieable(container[0], container[2])
False
```

1.2.10 auxjad.LoopByList

class auxjad.**LoopByList**(*contents: list*, ***, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: int = 0*, *move_window_on_first_call: bool = False*)

This class can be used to output slices of a `list` using the metaphor of a looping window of a variable size. This size is given by the argument `window_size`, which is an `int` representing how many elements are to be included in each slice.

For instance, if the initial container had the logical ties `[A, B, C, D, E, F]` (where each letter represents one elemnt of an arbitrary type) and the looping window was size `3`, the output would be:

```
A B C B C D C D E D E F E F F
```

This can be better visualised as:

```
A B C
  B C D
    C D E
      D E F
        E F
          F
```

It takes a list and the number of elements of the window as arguments. Each call of the object, in this case `looper()`, will move the window forwards and output the result:

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> looper()
['A', 'B', 'C']
>>> looper()
['B', 'C', 'D']
```

The property `current_window` can be used to access the current window without moving the head forwards.

```
>>> looper.current_window
['B', 'C', 'D']
```

The very first call will output the input list without processing it. To disable this behaviour and have the looping window move on the very first call, initialise the class with the keyword argument `move_window_on_first_call` set to `True`.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list,
...                            window_size=3,
...                            move_window_on_first_call=True,
...                            )
>>> looper()
['B', 'C', 'D']
```

The instances of `LoopByList` can also be used as an iterator, which can then be used in a for loop to exhaust all windows.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list,
...                            window_size=3,
...                            )
>>> for window in looper:
...     print(window)
['A', 'B', 'C']
['B', 'C', 'D']
['C', 'D', 'E']
['D', 'E', 'F']
['E', 'F']
['F']
```

This class can take many optional keyword arguments during its creation. `step_size` dictates the size of each individual step in number of elements (default value is 1). `max_steps` sets the maximum number of steps that the window can advance when the object is called, ranging between 1 and the input value (default is also 1). `repetition_chance` sets the chance of a window result repeating itself (that is, the window not moving forwards when called). It should range from 0.0 to 1.0 (default 0.0, i.e. no repetition). `forward_bias` sets the chance of the window moving forward instead of backwards. It should range from 0.0 to 1.0 (default 1.0, which means the window can only move forwards. A value of 0.5 gives 50% chance of moving forwards while a value of 0.0 will move the window only backwards). Finally, `head_position` can be used to offset the starting position of the looping window (default is 0).

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list,
...                            window_size=3,
...                            step_size=1,
...                            max_steps=2,
...                            repetition_chance=0.25,
...                            forward_bias=0.2,
...                            head_position=0,
...                            )
>>> looper.window_size
3
>>> looper.step_size
1
>>> looper.repetition_chance
0.25
>>> looper.forward_bias
0.2
>>> looper.max_steps
2
>>> looper.head_position
0
```

Use the properties below to change these values after initialisation.

```
>>> looper.window_size = 2
>>> looper.step_size = 2
>>> looper.max_steps = 3
>>> looper.repetition_chance = 0.1
>>> looper.forward_bias = 0.8
>>> looper.head_position = 2
>>> looper.window_size
2
>>> looper.step_size
2
```

(continues on next page)

(continued from previous page)

```
>>> looper.max_steps
3
>>> looper.repetition_chance
0.1
>>> looper.forward_bias
0.8
>>> looper.head_position
2
```

The function `len()` can be used to get the total number of elements in the container.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> len(looper)
6
```

To run through just part of the process and output it as a single list, starting from the initial head position, use the method `output_n()` and pass the number of iterations as argument.

```
>>> input_list = ['A', 'B', 'C', 'D']
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> looper.output_n(2)
['A', 'B', 'C', 'B', 'C', 'D']
```

To run through the whole process and output it as a single list, from the initial head position until the process outputs the single last element, use the method `output_all()`.

```
>>> input_list = ['A', 'B', 'C', 'D']
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> looper.output_all()
['A', 'B', 'C', 'B', 'C', 'D', 'C', 'D', 'D']
```

To change the size of the looping window after instantiation, use the property `window_size`. In the example below, the initial window is of size 3, and so the first call of the looper object outputs the first, second, and third elements of the list. The window size is then set to 4, and the looper is called again, moving to the element in the next position, thus outputting the second, third, fourth, and fifth elements.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> looper()
['A', 'B', 'C']
>>> looper.window_size = 4
>>> looper()
['B', 'C', 'D', 'E']
```

Use the `contents` property to read as well as overwrite the contents of the looper. Notice that the `head_position` will remain on its previous value and must be reset to `0` if that's required.

```
>>> input_list = ['A', 'B', 'C', 'D', 'E', 'F']
>>> looper = auxjad.LoopByList(input_list,
...                            window_size=3,
...                            )
>>> looper.contents
['A', 'B', 'C', 'D', 'E', 'F']
>>> looper()
['A', 'B', 'C']
```

(continues on next page)

(continued from previous page)

```
>>> looper()
['B', 'C', 'D']
>>> looper.contents = [0, 1, 2, 3, 4]
>>> looper.contents
[0, 1, 2, 3, 4]
>>> looper()
[2, 3, 4]
>>> looper.head_position = 0
>>> looper()
[0, 1, 2]
```

It should be clear that the list can contain any types of elements:

```
>>> input_list = [123, 'foo', (3, 4), 3.14]
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> looper()
[123, 'foo', (3, 4)]
```

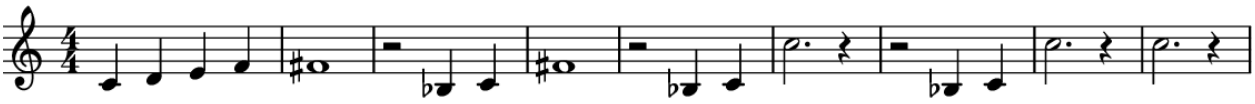
This also include Abjad's types. Abjad's exclusive membership requirement is respected since each call returns a `copy.deepcopy` of the window. The same is true to the `output_all()` method.

```
>>> import abjad
>>> import copy
>>> input_list = [
...     abjad.Container(r"c'4 d'4 e'4 f'4"),
...     abjad.Container(r"fs'1"),
...     abjad.Container(r"r2 bf4 c'4"),
...     abjad.Container(r"c''2. r4"),
... ]
>>> looper = auxjad.LoopByList(input_list, window_size=3)
>>> staff = abjad.Staff()
>>> for element in looper.output_all():
...     staff.append(element)
>>> abjad.f(staff)
\new Staff
{
    {
        c'4
        d'4
        e'4
        f'4
    }
    {
        fs'1
    }
    {
        r2
        bf4
        c'4
    }
    {
        fs'1
    }
    {
        r2
        bf4
```

(continues on next page)

(continued from previous page)

```
        c'4
    }
    {
        c''2.
        r4
    }
    {
        r2
        bf4
        c'4
    }
    {
        c''2.
        r4
    }
    {
        c''2.
        r4
    }
}
```



__init__(*contents: list*, ***, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: int = 0*, *move_window_on_first_call: bool = False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, window_size, ...)	Initialize self.
output_all()	Goes through the whole looping process and outputs a single list.
output_n(n)	Goes through n iterations of the looping process and outputs a single list.

Attributes

contents	The list which serves as the basis for the slices of the looper.
current_window	Read-only property, returns the window at the current head position.
forward_bias	The chance of the window moving forward instead of backwards.
head_position	The position of the head at the start of a looping window.
max_steps	The maximum number of steps per operation.

Continued on next page

Table 11 – continued from previous page

repetition_chance	The chance of the head not moving, thus repeating the output.
step_size	The size of each step when moving the head.
window_size	The length of the looping window.

1.2.11 auxjad.LoopByNotes

class auxjad.**LoopByNotes**(*contents: abjad.core.Container.Container*, ***, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: int = 0*, *omit_all_time_signatures: bool = False*, *force_identical_time_signatures: bool = False*, *move_window_on_first_call: bool = False*)

This class can be used to output slices of an `abjad.Container` using the metaphor of a looping window of a variable size. This size is given by the argument `window_size`, which is an `int` representing how many notes are to be included in each slice. The duration of the slice will be the sum of the duration of these notes.

For instance, if the initial container had the logical ties `[A, B, C, D, E, F]` (where each letter represents one logical tie) and the looping window was size `3`, the output would be:

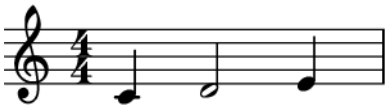
```
A B C B C D C D E D E F E F F
```

This can be better visualised as:

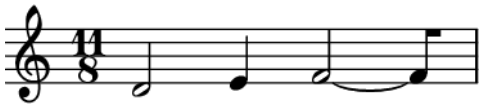
```
A B C
  B C D
    C D E
      D E F
        E F
          F
```

Usage is similar to other factory classes. It takes a container (or child class equivalent) and the number of elements of the window as arguments. Each call of the object, in this case `looper()`, will move the window forwards and output the result.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=3,
...                             )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
}
```

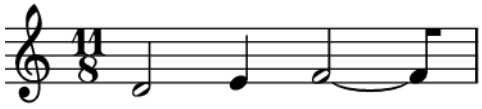


```
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 11/8
    d'2
    e'4
    f'2
    ~
    f'8
}
```



The property `current_window` can be used to access the current window without moving the head forwards.

```
>>> notes = looper.current_window
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 11/8
    d'2
    e'4
    f'2
    ~
    f'8
}
```



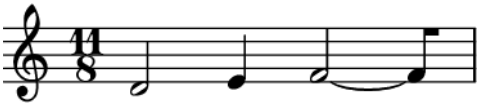
The very first call will output the input container without processing it. To disable this behaviour and have the looping window move on the very first call, initialise the class with the keyword argument `move_window_on_first_call` set to `True`.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(
...     input_music,
...     window_size=3,
...     move_window_on_first_call=True,
... )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 11/8
    d'2
    e'4
```

(continues on next page)

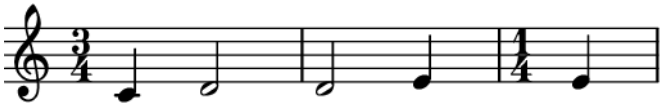
(continued from previous page)

```
    f'2
    ~
    f'8
}
```



The instances of `LoopByNotes` can also be used as an iterator, which can then be used in a for loop to exhaust all windows.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=2,
...                             )
>>> staff = abjad.Staff()
>>> for window in looper:
...     staff.append(window)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'2
    d'2
    e'4
    \time 1/4
    e'4
}
```



Notice how the second staff in the example above does not have a time signature. This is because consecutive identical time signatures are omitted by default. To change this behaviour, instantialise this class with the keyword argument `force_identical_time_signatures` set to `True`, or change the `force_identical_time_signatures` property to alter its value after the initialisation.

This class can take many optional keyword arguments during its creation. `step_size` dictates the size of each individual step in number of elements (default value is 1). `max_steps` sets the maximum number of steps that the window can advance when the object is called, ranging between 1 and the input value (default is also 1). `repetition_chance` sets the chance of a window result repeating itself (that is, the window not moving forwards when called). It should range from 0.0 to 1.0 (default 0.0, i.e. no repetition). `forward_bias` sets the chance of the window moving forward instead of backwards. It should range from 0.0 to 1.0 (default 1.0, which means the window can only move forwards. A value of 0.5 gives 50% chance of moving forwards while a value of 0.0 will move the window only backwards). Finally, `head_position` can be used to offset the starting position of the looping window. It must be an integer and its default value is 0.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=3,
```

(continues on next page)

(continued from previous page)

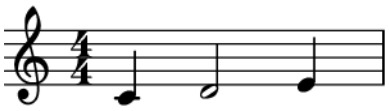
```
...                                 step_size=1,
...                                 max_steps=2,
...                                 repetition_chance=0.25,
...                                 forward_bias=0.2,
...                                 head_position=0,
...                                 omit_all_time_signatures=False,
...                                 force_identical_time_signatures=False,
...                                 )
>>> looper.window_size
3
>>> looper.step_size
1
>>> looper.repetition_chance
0.25
>>> looper.forward_bias
0.2
>>> looper.max_steps
2
>>> looper.head_position
0
>>> looper.omit_all_time_signatures
False
>>> looper.force_identical_time_signatures
False
```

Use the properties below to change these values after initialisation.

```
>>> looper.window_size = 2
>>> looper.step_size = 2
>>> looper.max_steps = 3
>>> looper.repetition_chance = 0.1
>>> looper.forward_bias = 0.8
>>> looper.head_position = 2
>>> looper.omit_all_time_signatures = True
>>> looper.force_identical_time_signatures = True
>>> looper.window_size
2
>>> looper.step_size
2
>>> looper.max_steps
3
>>> looper.repetition_chance
0.1
>>> looper.forward_bias
0.8
>>> looper.head_position
2
>>> looper.omit_all_time_signatures
True
>>> looper.force_identical_time_signatures
True
```

To disable time signatures altogether, initialise `LoopByNotes` with the keyword argument `omit_all_time_signatures` set to `True` (default is `False`), or use the `omit_time_signature` property after initialisation.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=3,
...                             omit_all_time_signatures=True,
...                             )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    d'2
    e'4
}
```

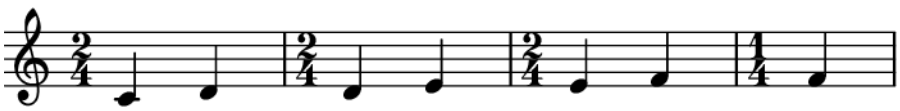


The function `len()` can be used to get the total number of elements in the contents.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=3,
...                             )
>>> len(looper)
5
```

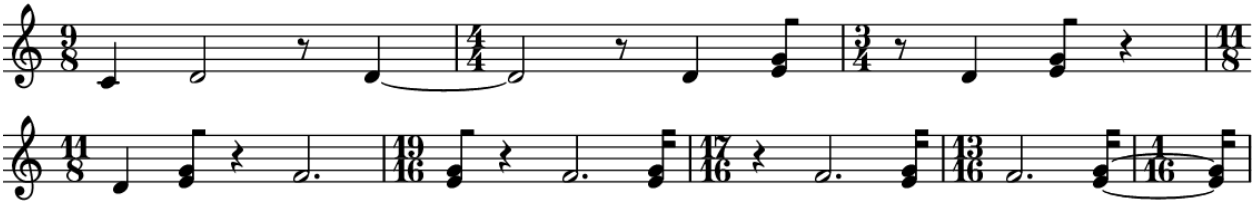
To run through the whole process and output it as a single container, from the initial head position until the process outputs the single last element, use the method `output_all()`.

```
>>> input_music = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=2,
...                             )
>>> window = looper.output_all()
>>> staff = abjad.Staff(window)
>>> abjad.f(staff)
\new Staff
{
    \time 2/4
    c'4
    d'4
    \time 2/4
    d'4
    e'4
    \time 2/4
    e'4
    f'4
    \time 1/4
    f'4
}
```



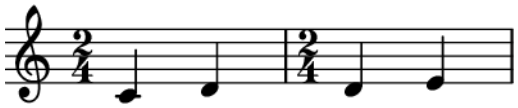
When using `output_all()`, set the keyword argument `tie_identical_pitches` to `True` in order to tie identical notes or chords at the end and beginning of consecutive windows.

```
>>> input_music = abjad.Container(r"c'4 d'2 r8 d'4 <e' g'>8 r4 f'2. "
...                               "<e' g'>16")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=4,
...                             )
>>> music = looper.output_all(tie_identical_pitches=True)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 9/8
    c'4
    d'2
    r8
    d'4
    ~
    \time 4/4
    d'2
    r8
    d'4
    <e' g'>8
    \time 3/4
    r8
    d'4
    <e' g'>8
    r4
    \time 11/8
    d'4
    <e' g'>8
    r4
    f'2.
    \time 19/16
    <e' g'>8
    r4
    f'2.
    <e' g'>16
    \time 17/16
    r4
    f'2.
    <e' g'>16
    \time 13/16
    f'2.
    <e' g'>16
    ~
    \time 1/16
    <e' g'>16
}
```



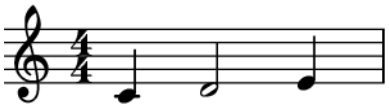
To run through just part of the process and output it as a single container, starting from the initial head position, use the method `output_n()` and pass the number of iterations as argument. Similarly to `output_all()`, the keyword argument `tie_identical_pitches` is available for tying pitches.

```
>>> input_music = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=2,
...                             )
>>> window = looper.output_n(2)
>>> staff = abjad.Staff(window)
>>> abjad.f(staff)
\new Staff
{
    \time 2/4
    c'4
    d'4
    \time 2/4
    d'4
    e'4
}
```



To change the size of the looping window after instantiation, use the property `window_size`. In the example below, the initial window is of size 3, and so the first call of the looper object outputs the first, second, and third leaves. The window size is then set to 4, and the looper is called again, moving to the leaf in the next position, thus outputting the second, third, fourth, and fifth leaves.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=3,
...                             )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
}
```

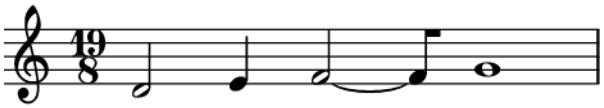


```
>>> looper.window_size = 4
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
```

(continues on next page)

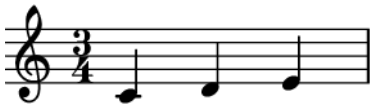
(continued from previous page)

```
{
    \time 19/8
    d'2
    e'4
    f'2
    ~
    f'8
    g'1
}
```

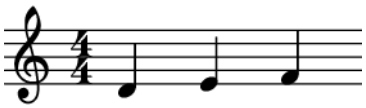


Use the `contents` property to read as well as overwrite the contents of the looper. Notice that the `head_position` will remain on its previous value and must be reset to `0` if that's required.

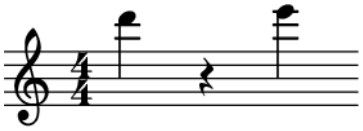
```
>>> input_music = abjad.Container(r"c'4 d'4 e'4 f'4 g'4 a'4")
>>> looper = auxjad.LoopByNotes(input_music,
>>>                             window_size=3,
>>>                             )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'4
    e'4
}
```



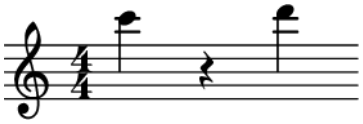
```
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    d'4
    e'4
    f'4
}
```



```
>>> looper.contents = abjad.Container(r"c'''4 r4 d'''4 r4 "
...                                    "e'''4 r4 f'''4 r4")
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    d'''4
    r4
    e'''4
}
```



```
>>> looper.head_position = 0
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'''4
    r4
    d'''4
}
```



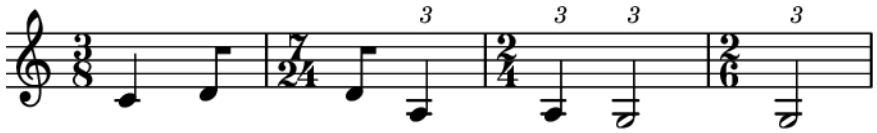
This class can handle tuplets, but the output is not ideal and so this functionality should be considered experimental. Time signatures will be correct when dealing with partial tuplets (thus having non-standard values in their denominators), but each individual note of a tuplet will have the ratio printed above it.

```
>>> input_music = abjad.Container(r"c'4 d'8 \times 2/3 {a4 g2}")
>>> looper = auxjad.LoopByNotes(input_music,
...                             window_size=2,
...                             )
>>> window = looper.output_all()
>>> staff = abjad.Staff(window)
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    d'8
    #(ly:expect-warning "strange time signature found")
    \time 7/24
    d'8
    \tweak edge-height #'(0.7 . 0)
    \times 2/3 {
```

(continues on next page)

(continued from previous page)

```
        a4
    }
    \tweak edge-height #'(0.7 . 0)
    \times 2/3 {
        \time 2/4
        a4
    }
    \tweak edge-height #'(0.7 . 0)
    \times 2/3 {
        g2
    }
    \tweak edge-height #'(0.7 . 0)
    \times 2/3 {
        #(ly:expect-warning "strange time signature found")
        \time 2/6
        g2
    }
}
```



__init__(*contents: abjad.core.Container.Container*, *, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: int = 0*, *omit_all_time_signatures: bool = False*, *force_identical_time_signatures: bool = False*, *move_window_on_first_call: bool = False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, window_size, ...)	Initialize self.
output_all(*, tie_identical_pitches)	Goes through the whole looping process and outputs a single abjad.Selection.
output_n(n, *, tie_identical_pitches)	Goes through n iterations of the looping process and outputs a single abjad.Selection.

Attributes

contents	The list which serves as the basis for the slices of the looper.
current_window	Read-only property, returns the window at the current head position.
force_identical_time_signatures	When True, identical time signatures will not be removed from the output.
forward_bias	The chance of the window moving forward instead of backwards.
head_position	The position of the head at the start of a looping window.

Continued on next page

Table 13 – continued from previous page

max_steps	The maximum number of steps per operation.
omit_all_time_signatures	When True, the output will contain no time signatures.
repetition_chance	The chance of the head not moving, thus repeating the output.
step_size	The size of each step when moving the head.
window_size	The length of the looping window.

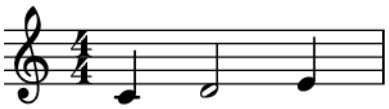
1.2.12 auxjad.LoopByWindow

class auxjad.**LoopByWindow**(*contents: abjad.core.Container.Container*, *, *window_size: (<class 'tuple'>, <class 'abjad.meter.Meter'>) = (4, 4)*, *step_size: (<class 'int'>, <class 'float'>, <class 'tuple'>, <class 'str'>, <class 'abjad.utilities.Duration.Duration'>) = (1, 16)*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: (<class 'int'>, <class 'float'>, <class 'tuple'>, <class 'str'>, <class 'abjad.utilities.Duration.Duration'>) = 0*, *omit_time_signature: bool = False*, *move_window_on_first_call: bool = False*)

This class can be used to output slices of an abjad.Container using the metaphor of a looping window of a constant size given by an abjad.Duration.

Usage is similar to other factory classes. It takes a container (or child class equivalent) as argument. Each call of the object, in this case looper(), will move the window forwards and output the sliced window. If no window_size nor step_size are entered as arguments, they are set to the following default values, respectively: (4, 4), i.e. a window of the size of a 4/4 bar, and (1, 16), i.e. a step of the length of a sixteenth-note.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music)
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
}
```

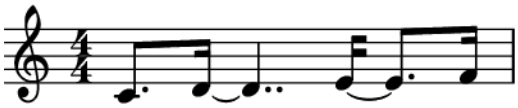


```
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'8.
    d'16
    ~
    d'4..
    e'16
```

(continues on next page)

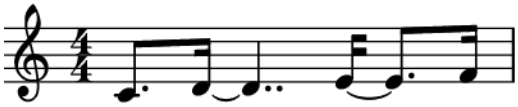
(continued from previous page)

```
    ~
    e'8.
    f'16
}
```



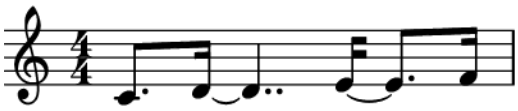
The property `current_window` can be used to access the current window without moving the head forwards.

```
>>> notes = looper.current_window()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'8.
    d'16
    ~
    d'4..
    e'16
    ~
    e'8.
    f'16
}
```



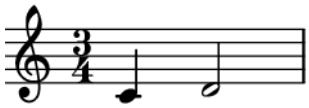
The very first call will output the input container without processing it. To disable this behaviour and have the looping window move on the very first call, initialise the class with the keyword argument `move_window_on_first_call` set to `True`.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music,
...                              move_window_on_first_call=True,
...                              )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'8.
    d'16
    ~
    d'4..
    e'16
    ~
    e'8.
    f'16
}
```

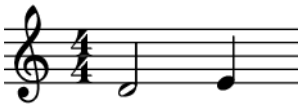


The optional arguments `window_size` and `step_size` can be used to set different window and step sizes. `window_size` can take a tuple or an `abjad.Meter` as input, while `step_size` takes a tuple or an `abjad.Duration`.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 4),
...                              )
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'2
}
```



```
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    d'2
    e'4
}
```



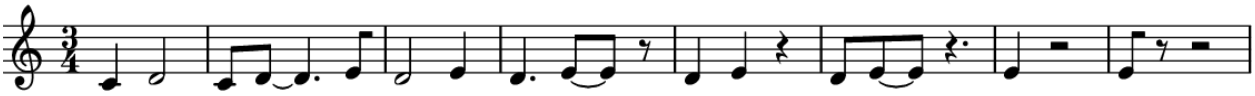
The instances of `LoopByWindow` can also be used as an iterator, which can then be used in a for loop to exhaust all windows. Notice how it appends rests at the end of the container, until it is totally exhausted.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 8),
...                              )
>>> staff = abjad.Staff()
>>> for window in looper:
...     staff.append(window)
>>> abjad.f(staff)
\new Staff
```

(continues on next page)

(continued from previous page)

```
{
    \time 3/4
    c'4
    d'2
    c'8
    d'8
    ~
    d'4.
    e'8
    d'2
    e'4
    d'4.
    e'8
    ~
    e'8
    r8
    d'4
    e'4
    r4
    d'8
    e'8
    ~
    e'8
    r4.
    e'4
    r2
    e'8
    r8
    r2
}
```



This class can take many optional keyword arguments during its creation, besides `window_size` and `step_size`. `max_steps` sets the maximum number of steps that the window can advance when the object is called, ranging between 1 and the input value (default is also 1). `repetition_chance` sets the chance of a window result repeating itself (that is, the window not moving forwards when called). It should range from 0.0 to 1.0 (default 0.0, i.e. no repetition). `forward_bias` sets the chance of the window moving forward instead of backwards. It should range from 0.0 to 1.0 (default 1.0, which means the window can only move forwards. A value of 0.5 gives 50% chance of moving forwards while a value of 0.0 will move the window only backwards). Finally, `head_position` can be used to offset the starting position of the looping window. It must be a tuple or an `abjad.Duration`, and its default value is 0.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(5, 8),
...                              max_steps=2,
...                              repetition_chance=0.25,
...                              forward_bias=0.2,
...                              head_position=(2, 8),
...                              omit_time_signature=False,
...                              )
```

(continues on next page)

(continued from previous page)

```
>>> looper.window_size
3/4
>>> looper.step_size
5/8
>>> looper.repetition_chance
0.25
>>> looper.forward_bias
0.2
>>> looper.max_steps
2
>>> looper.head_position
1/4
>>> looper.omit_time_signature
False
```

Use the properties below to change these values after initialisation.

```
>>> looper.window_size = (5, 4)
>>> looper.step_size = (1, 4)
>>> looper.max_steps = 3
>>> looper.repetition_chance = 0.1
>>> looper.forward_bias = 0.8
>>> looper.head_position = 0
>>> looper.omit_time_signature = True
>>> looper.window_size
5/4
>>> looper.step_size
1/4
>>> looper.max_steps
3
>>> looper.repetition_chance
0.1
>>> looper.forward_bias
0.8
>>> looper.head_position
0
>>> looper.omit_time_signature
True
```

The function `len()` can be used to get the total number of steps in the contents (always rounded up).

```
>>> input_music = abjad.Container(r"c'1")
>>> looper = auxjad.LoopByWindow(input_music)
>>> len(looper)
16
>>> input_music = abjad.Container(r"c'1")
>>> looper = auxjad.LoopByWindow(input_music,
...                              step_size=(1, 4),
...                              )
>>> len(looper)
4
>>> input_music = abjad.Container(r"c'2..")
>>> looper = auxjad.LoopByWindow(input_music,
...                              step_size=(1, 4),
...                              window_size=(2, 4),
...                              )
>>> len(looper)
```

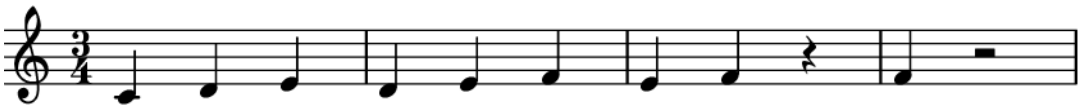
(continues on next page)

(continued from previous page)

```
4
```

To run through the whole process and output it as a single container, from the initial head position until the process outputs the single last element, use the method `output_all()`.

```
>>> input_music = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 4),
...                              )
>>> music = looper.output_all()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'4
    e'4
    d'4
    e'4
    f'4
    e'4
    f'4
    r4
    f'4
    r2
}
```



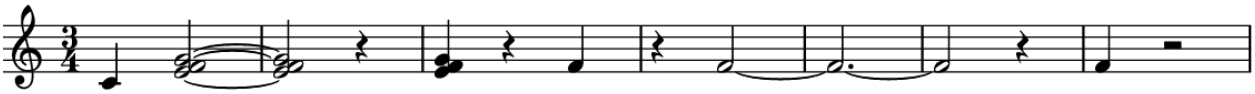
When using `output_all()`, set the keyword argument `tie_identical_pitches` to `True` in order to tie identical notes or chords at the end and beginning of consecutive windows.

```
>>> input_music = abjad.Container(r"c'4 <e' f' g'>2 r4 f'2.")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 4),
...                              )
>>> music = looper.output_all(tie_identical_pitches=True)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    <e' f' g'>2
    ~
    <e' f' g'>2
    r4
    <e' f' g'>4
    r4
    f'4
```

(continues on next page)

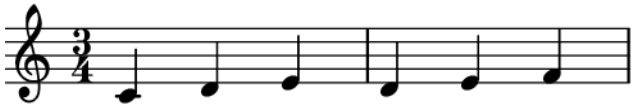
(continued from previous page)

```
    r4
    f'2
    ~
    f'2.
    ~
    f'2
    r4
    f'4
    r2
}
```



To run through just part of the process and output it as a single container, starting from the initial head position, use the method `output_n()` and pass the number of iterations as argument. Similarly to `output_all()`, the keyword argument `tie_identical_pitches` is available for tying pitches.

```
>>> input_music = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 4),
...                              )
>>> music = looper.output_n(2)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'4
    e'4
    d'4
    e'4
    f'4
}
```



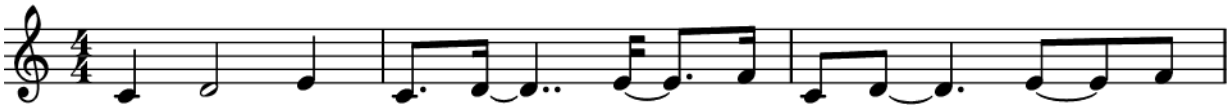
To change the size of the looping window after instantiation, use the property `window_size`. In the example below, the initial window is of size (4, 4), but changes to (3, 8) after three calls. Notice how the very first call attaches a time signature equivalent to the window size to the output window; subsequent calls will not have time signatures unless the size of the looping window changes.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music)
>>> staff = abjad.Staff()
>>> for _ in range(3):
...     notes = looper()
...     staff.append(notes)
```

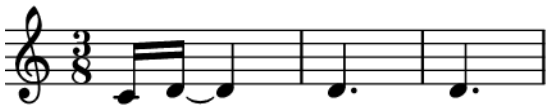
(continues on next page)

(continued from previous page)

```
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
    c'8.
    d'16
    ~
    d'4..
    e'16
    ~
    e'8.
    f'16
    c'8
    d'8
    ~
    d'4.
    e'8
    ~
    e'8
    f'8
}
```

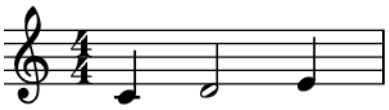


```
>>> looper.window_size = (3, 8)
>>> staff = abjad.Staff()
>>> for _ in range(3):
...     notes = looper()
...     staff.append(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'16
    d'16
    ~
    d'4
    d'4.
    d'4.
}
```



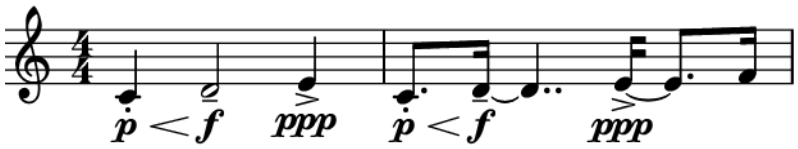
To disable time signatures altogether, initialise `LoopByWindow` with the keyword argument `omit_time_signature` set to `True` (default is `False`), or use the `omit_time_signature` property after initialisation.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music, omit_time_signature=True)
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'4
    d'2
    e'4
}
```



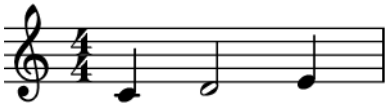
This class can handle dynamics and articulations too. When a leaf is shortened by the looping window's movement, the dynamics and articulations are still applied to it.

```
>>> input_music = abjad.Container(
... r"c'4-.\p\< d'2--\f e'4->\ppp f'2 ~ f'8")
>>> looper = auxjad.LoopByWindow(input_music)
>>> staff = abjad.Staff()
>>> for _ in range(2):
...     music = looper()
...     staff.append(music)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    \p
    - \staccato
    \<
    d'2
    \f
    - \tenuto
    e'4
    \ppp
    - \accent
    c'8.
    \p
    - \staccato
    \<
    d'16
    \f
    - \tenuto
    ~
    d'4..
    e'16
    \ppp
    - \accent
    ~
    e'8.
    f'16
}
```

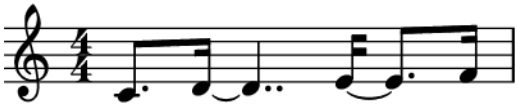


Use the `contents` property to read as well as overwrite the contents of the looper. Notice that the `head_position` will remain on its previous value and must be reset to `0` if that's required.

```
>>> input_music = abjad.Container(r"c'4 d'2 e'4 f'2 ~ f'8 g'1")
>>> looper = auxjad.LoopByWindow(input_music)
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
}
```



```
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'8.
    d'16
    ~
    d'4..
    e'16
    ~
    e'8.
    f'16
}
```

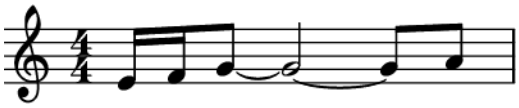


```
>>> looper.contents = abjad.Container(r"c'16 d'16 e'16 f'16 g'2. a'1")
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    e'16
    f'16
```

(continues on next page)

(continued from previous page)

```
    g'8
    ~
    g'2
    ~
    g'8
    a'8
}
```



```
>>> looper.head_position = 0
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    c'16
    d'16
    e'16
    f'16
    g'2.
}
```

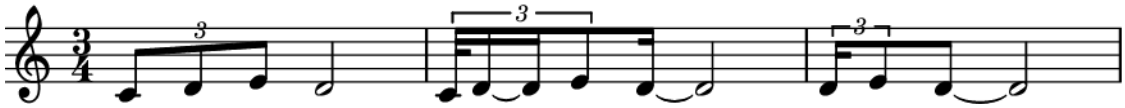
This class can handle tuplets, but this functionality should be considered experimental.

```
>>> input_music = abjad.Container(r"\times 2/3 {c'8 d'8 e'} d'2.")
>>> looper = auxjad.LoopByWindow(input_music,
...                              window_size=(3, 4),
...                              step_size=(1, 16))
>>> staff = abjad.Staff()
>>> for _ in range(3):
...     window = looper()
...     staff.append(window)
>>> abjad.f(staff)
\new Staff
{
    \times 2/3 {
        \time 3/4
        c'8
        d'8
        e'8
    }
    d'2
    \times 2/3 {
        c'32
        d'16
        ~
```

(continues on next page)

(continued from previous page)

```
        d'16
        e'8
    }
    d'16
    ~
    d'2
    \times 2/3 {
        d'16
        e'8
    }
    d'8
    ~
    d'2
}
```



__init__(*contents: abjad.core.Container.Container*, ***, *window_size: (<class 'tuple'>, <class 'abjad.meter.Meter'>) = (4, 4)*, *step_size: (<class 'int'>, <class 'float'>, <class 'tuple'>, <class 'str'>, <class 'abjad.utilities.Duration.Duration'>) = (1, 16)*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *forward_bias: float = 1.0*, *head_position: (<class 'int'>, <class 'float'>, <class 'tuple'>, <class 'str'>, <class 'abjad.utilities.Duration.Duration'>) = 0*, *omit_time_signature: bool = False*, *move_window_on_first_call: bool = False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, window_size, ...)	Initialize self.
output_all(*, tie_identical_pitches)	Goes through the whole looping process and outputs a single abjad.Selection.
output_n(n, *, tie_identical_pitches)	Goes through n iterations of the looping process and outputs a single abjad.Selection.

Attributes

contents	The list which serves as the basis for the slices of the looper.
current_window	Read-only property, returns the window at the current head position.
forward_bias	The chance of the window moving forward instead of backwards.
head_position	The position of the head at the start of a looping window.
max_steps	The maximum number of steps per operation.
omit_time_signature	When True, the output will contain no time signatures.

Continued on next page

Table 15 – continued from previous page

repetition_chance	The chance of the head not moving, thus repeating the output.
step_size	The size of each step when moving the head.
window_size	The length of the looping window.

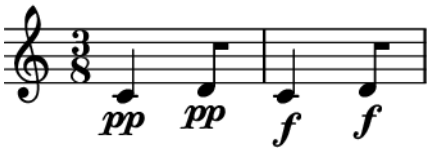
1.2.13 auxjad.remove_repeated_dynamics

auxjad.**remove_repeated_dynamics**(*container: abjad.core.Container.Container*, *, *ignore_hairpins: bool = False*, *reset_after_rests: bool = False*)

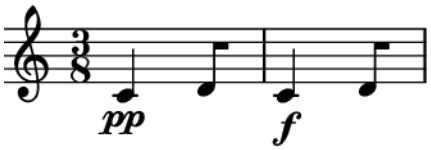
Mutates an input container (of type abjad.Container or child class) in place and has no return value. This function removes all consecutive repeated dynamic markings.

When two consecutive leaves have identical dynamics, the second one is removed:

```
>>> staff = abjad.Staff(r"\time 3/8 c'4\pp d'8\pp | c'4\f d'8\f")
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    \pp
    d'8
    \pp
    c'4
    \f
    d'8
    \f
}
```

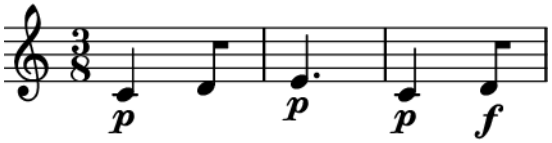



```
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    \pp
    d'8
    c'4
    \f
    d'8
}
```

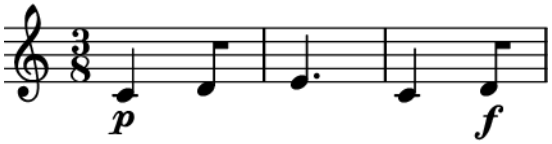


The function also removes dynamics that are separated by an arbitrary number of leaves without dynamics:

```
>>> staff = abjad.Staff(r"\time 3/8 c'4\p d'8 | e'4.\p | c'4\p d'8\f")
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    \p
    d'8
    e'4.
    \p
    c'4
    \p
    d'8
    \f
}
```



```
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    \p
    d'8
    e'4.
    c'4
    d'8
    \f
}
```



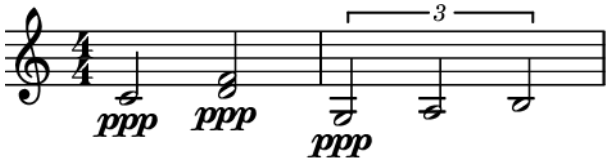
The input container can also handle subcontainers:

```
>>> staff = abjad.Staff([abjad.Note("c'2"),
...                      abjad.Chord("<d' f'>2"),
...                      abjad.Tuplet((2, 3), "g2 a2 b2"),
...                      ])
>>> abjad.attach(abjad.Dynamic('ppp'), staff[0])
>>> abjad.attach(abjad.Dynamic('ppp'), staff[1])
>>> abjad.attach(abjad.Dynamic('ppp'), staff[2][0])
>>> abjad.f(staff)
\new Staff
{
```

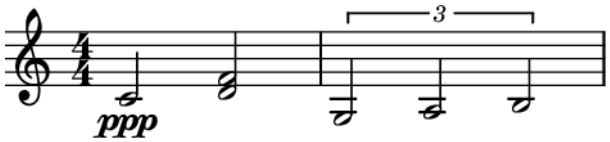
(continues on next page)

(continued from previous page)

```
    c'2
    \ppp
    <d' f'>2
    \ppp
    \times 2/3 {
        g2
        \ppp
        a2
        b2
    }
}
```




```
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    c'2
    \ppp
    <d' f'>2
    \times 2/3 {
        g2
        a2
        b2
    }
}
```



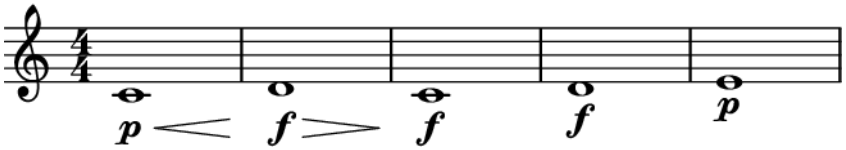
By default, repeated dynamics with hairpins in between are not removed, but consecutive ones will.

```
>>> staff = abjad.Staff(r"c'2\p\< d'2\f\> | c'2\f d'2\f | e'1\p")
>>> abjad.f(staff)
\new Staff
{
    c'1
    \p
    \<
    d'1
    \f
    \>
    c'1
    \f
    d'1
    \f
```

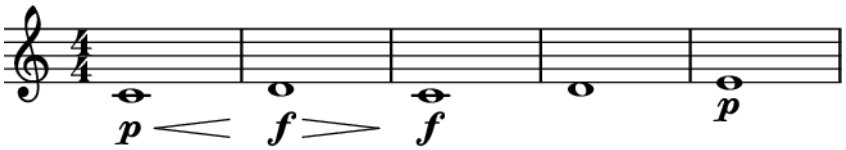
(continues on next page)

(continued from previous page)

```
    e'1
    \p
}
```

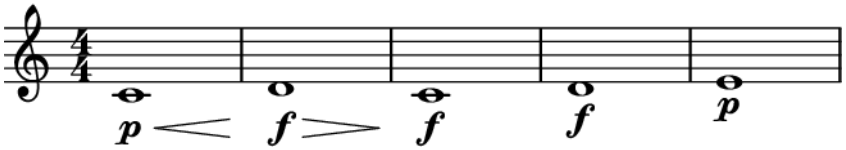


```
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    c'1
    \p
    \<
    d'1
    \f
    \>
    c'1
    \f
    d'1
    e'1
    \p
}
```

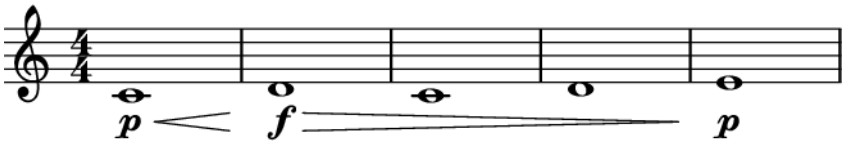


To override the previous behaviour, set `ignore_hairpins=True` and hairpins will be ignored.

```
>>> staff = abjad.Staff(r"c'2\p\< d'2\f\> | c'2\f d'2\f | e'1\p")
>>> abjad.f(staff)
\new Staff
{
    c'1
    \p
    \<
    d'1
    \f
    \>
    c'1
    \f
    d'1
    \f
    e'1
    \p
}
```

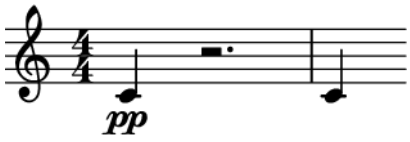


```
>>> auxjad.remove_repeated_dynamics(staff, ignore_hairpins=True)
>>> abjad.f(staff)
\new Staff
{
    c'1
    \p
    \<
    d'1
    \f
    \>
    c'1
    d'1
    e'1
    \p
}
```



By default, rests are treated just like any other leaf and thus notes with an identical dynamic separated by an arbitrary number of rests will be considered as repeated and the second dynamic will be removed.

```
>>> staff = abjad.Staff(r"c'4\pp r2. | c'4\pp")
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    r2.
    c'4
}
```



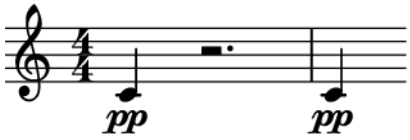
To override the previous behaviour, set `reset_after_rests=True` and dynamics will always be restated after a rest.

```
>>> staff = abjad.Staff(r"c'4\pp r2. | c'4\pp")
>>> auxjad.remove_repeated_dynamics(staff, reset_after_rests=True)
>>> abjad.f(staff)
\new Staff
```

(continues on next page)

(continued from previous page)

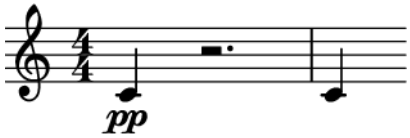
```
{
    c'4
    \pp
    r2.
    c'4
    \pp
}
```



The argument `reset_after_rests` takes not only boolean values but also duration (`abjad.Duration`, tuple, float, etc.). This sets the maximum length of rests before which identical dynamics are restated. If the total length of rests falls below that value, then repeated dynamics are removed.

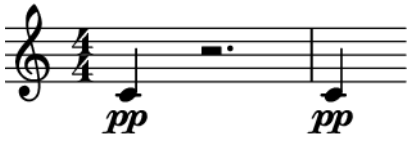
In the case below, a rest of `r2.` is shorter than a duration of (4, 4), so the repeated dynamic is removed.

```
>>> staff = abjad.Staff(r"c'4\pp r2. | c'4\pp")
>>> auxjad.remove_repeated_dynamics(staff, reset_after_rests=(4, 4))
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    r2.
    c'4
}
```



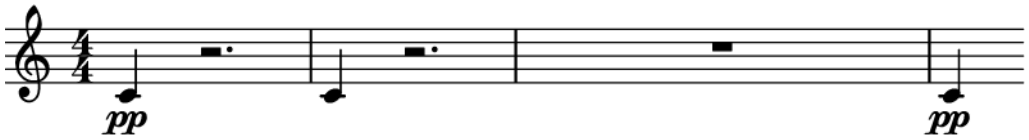
But setting the duration to 2/4 forces the dynamic to be restated.

```
>>> staff = abjad.Staff(r"c'4\pp r2. | c'4\pp")
>>> auxjad.remove_repeated_dynamics(staff, reset_after_rests=2/4)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    r2.
    c'4
    \pp
}
```



The function also handles measure rests with `reset_after_rests`.

```
>>> staff = abjad.Staff(r"c'4\pp r2. | c'4\pp r2. |R1 | c'4\pp")
>>> auxjad.remove_repeated_dynamics(
...     staff,
...     reset_after_rests=abjad.Duration(4, 4),
... )
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    r2.
    c'4
    r2.
    R1
    c'4
    \pp
}
```



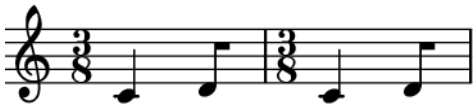
1.2.14 auxjad.remove_repeated_time_signatures

auxjad.**remove_repeated_time_signatures**(*container: abjad.core.Container.Container*)

Mutates an input container (of type `abjad.Container` or child class) in place and has no return value. This function removes all consecutive repeated time signatures.

When two consecutive bars have identical time signatures, the second one is removed:

```
>>> staff = abjad.Staff(r"c'4 d'8 | c'4 d'8")
>>> abjad.attach(abjad.TimeSignature((3, 8)), staff[0])
>>> abjad.attach(abjad.TimeSignature((3, 8)), staff[2])
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    d'8
    \time 3/8
    c'4
    d'8
}
```

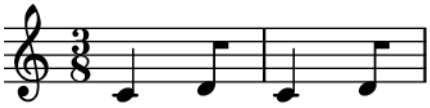


```
>>> auxjad.remove_repeated_time_signatures(staff)
>>> abjad.f(staff)
\new Staff
```

(continues on next page)

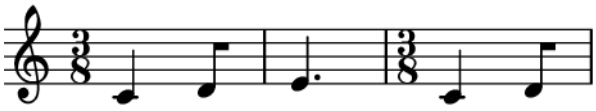
(continued from previous page)

```
{
    \time 3/8
    c'4
    d'8
    c'4
    d'8
}
```

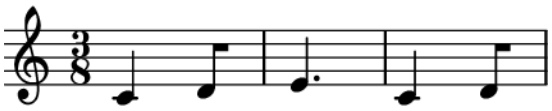


The function also removes time signatures that are separated by an arbitrary number of bars without one:

```
>>> staff = abjad.Staff(r"c'4 d'8 e'4. c'4 d'8")
>>> abjad.attach(abjad.TimeSignature((3, 8)), staff[0])
>>> abjad.attach(abjad.TimeSignature((3, 8)), staff[3])
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    d'8
    e'4.
    \time 3/8
    c'4
    d'8
}
```

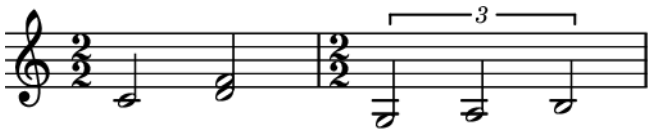


```
>>> auxjad.remove_repeated_time_signatures(staff)
>>> abjad.f(staff)
\new Staff
{
    \time 3/8
    c'4
    d'8
    e'4.
    c'4
    d'8
}
```

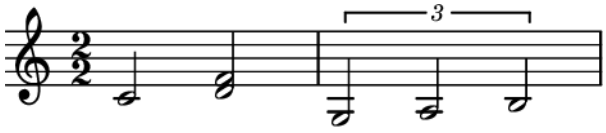


The input container can also handle subcontainers, including cases in which the time signatures are attached to leaves of subcontainers:

```
>>> staff = abjad.Staff([abjad.Note("c'2"),
...                      abjad.Chord("<d' f'>2"),
...                      abjad.Tuplet((2, 3), "g2 a2 b2"),
...                      ])
>>> abjad.attach(abjad.TimeSignature((2, 2)), staff[0])
>>> abjad.attach(abjad.TimeSignature((2, 2)), staff[2][0])
>>> abjad.f(staff)
\new Staff
{
    \time 2/2
    c'2
    <d' f'>2
    \times 2/3 {
        \time 2/2
        g2
        a2
        b2
    }
}
```




```
>>> auxjad.remove_repeated_time_signatures(staff)
>>> abjad.f(staff)
\new Staff
{
    \time 2/2
    c'2
    <d' f'>2
    \times 2/3 {
        g2
        a2
        b2
    }
}
```



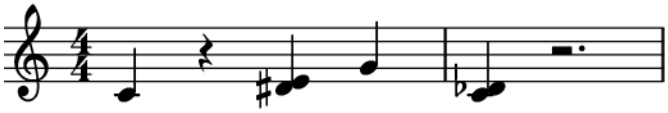
1.2.15 auxjad.respell_container

auxjad.**respell_container**(*container: abjad.core.Container.Container*, *, *include_multiples: bool = False*, *respell_by_pitch_class: bool = False*)

Mutates an input container (of type `abjad.Container` or child class) in place and has no return value. This function changes the accidentals of individual pitches of all chords in a container in order to avoid augmented unisons.

To use this function, apply it to a container that contains chords with augmented unisons.

```
>>> container = abjad.Container(r"c'4 r4 <ef' e'>4 g'4 <c' cs'>4 r2.")
>>> auxjad.respell_container(container)
>>> abjad.f(container)
{
    c'4
    r4
    <ds' e'>4
    g'4
    <c' df'>4
    r2.
}
```



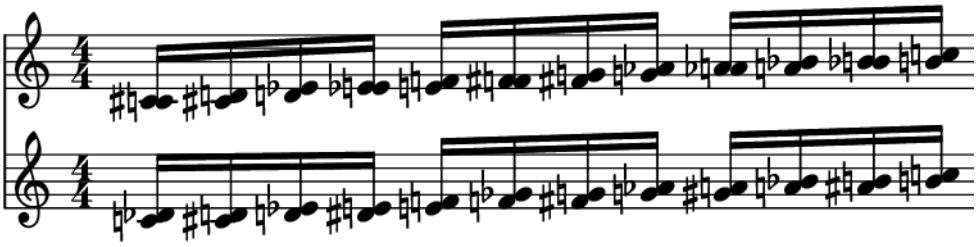
The example below shows the default spelling of 2-note chords by `Abjad` in the upper staff, and the respelt 2-note chords in the bottom staff.

```
>>> staff1 = abjad.Staff()
>>> staff2 = abjad.Staff()
>>> for pitch in range(12):
...     staff1.append(abjad.Chord([pitch, pitch + 1], (1, 16)))
...     staff2.append(abjad.Chord([pitch, pitch + 1], (1, 16)))
>>> auxjad.respell_container(staff2)
>>> literal = abjad.LilyPondLiteral(r'\accidentalStyle dodecaphonic')
>>> abjad.attach(literal, staff1)
>>> abjad.attach(literal, staff2)
>>> score = abjad.Score([staff1, staff2])
>>> abjad.f(score)
\new Score
<<
    \new Staff
    {
        \accidentalStyle dodecaphonic
        <c' cs'>16
        <cs' d'>16
        <d' ef'>16
        <ef' e'>16
        <e' f'>16
        <f' fs'>16
        <fs' g'>16
        <g' af'>16
        <af' a'>16
        <a' bf'>16
        <bf' b'>16
        <b' c''>16
    }
    \new Staff
    {
        \accidentalStyle dodecaphonic
        <c' df'>16
        <cs' d'>16
        <d' ef'>16
        <ds' e'>16
        <e' f'>16
```

(continues on next page)

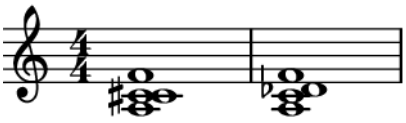
(continued from previous page)

```
        <f' gf'>16
        <fs' g'>16
        <g' af'>16
        <gs' a'>16
        <a' bf'>16
        <as' b'>16
        <b' c''>16
    }
>>
```



The function looks for all augmented unissons in chords of 3 or more pitches:

```
>>> container1 = abjad.Container(r"<a c' cs' f'>1")
>>> container2 = abjad.Container(r"<a c' cs' f'>1")
>>> auxjad.respell_container(container2)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <a c' cs' f'>1
    }
    {
        <a c' df' f'>1
    }
}
```



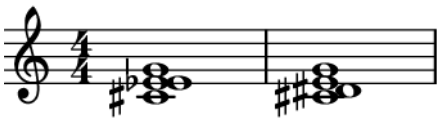
It is not a problem if the pitches are input out of order.

```
>>> container1 = abjad.Container(r"<e' cs' g' ef'>1")
>>> container2 = abjad.Container(r"<e' cs' g' ef'>1")
>>> auxjad.respell_container(container2)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <cs' ef' e' g'>1
    }
    {
```

(continues on next page)

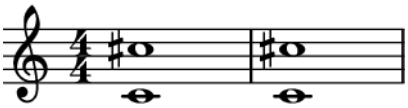
(continued from previous page)

```
        <cs' ds' e' g'>1
    }
}
```



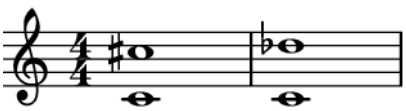
By default, this function only changes spelling for pitches that are 1 semitone apart.

```
>>> container1 = abjad.Container(r"<c' cs''>1")
>>> container2 = abjad.Container(r"<c' cs''>1")
>>> auxjad.respell_container(container2)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <c' cs''>1
    }
    {
        <c' cs''>1
    }
}
```



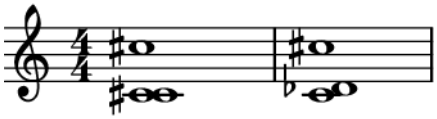
To consider pitches in different octaves (thus including augmented unisons, augmented octaves, augmented fifteenths, etc.), call this function with the keyword argument `include_multiples` set to `True`.

```
>>> container1 = abjad.Container(r"<c' cs''>1")
>>> container2 = abjad.Container(r"<c' cs''>1")
>>> auxjad.respell_container(container2, include_multiples=True)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <c' cs''>1
    }
    {
        <c' df''>1
    }
}
```



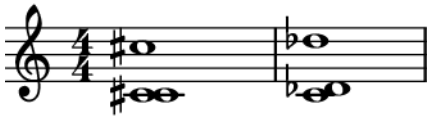
By default, when this function changes the spelling of a pitch, it does not change the spelling of all other pitches with the same pitch-class.

```
>>> container1 = abjad.Container(r"<c' cs' cs''>1")
>>> container2 = abjad.Container(r"<c' cs' cs''>1")
>>> auxjad.respell_container(container2)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <c' cs' cs''>1
    }
    {
        <c' df' cs''>1
    }
}
```



To alter all pitch-classes, call this function with the keyword argument `respell_by_pitch_class` set to `True`.

```
>>> container1 = abjad.Container(r"<c' cs' cs''>1")
>>> container2 = abjad.Container(r"<c' cs' cs''>1")
>>> auxjad.respell_container(container2, respell_by_pitch_class=True)
>>> staff = abjad.Staff([container1, container2])
>>> abjad.f(staff)
\new Staff
{
    {
        <c' cs' cs''>1
    }
    {
        <c' df' df''>1
    }
}
```



1.2.16 auxjad.respell_chord

auxjad.**respell_chord**(*chord: abjad.core.Chord.Chord*, *, *include_multiples: bool = False*, *respell_by_pitch_class: bool = False*)

Mutates an input chord (of type `abjad.Chord` or child class) in place and has no return value. This function changes the accidentals of individual pitches of a chord in order to avoid augmented unisons.

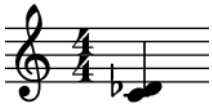
To use this function, apply it to a chord that contains augmented unisons.

```
>>> chord = abjad.Chord("<c' cs'>4")
>>> auxjad.respell_chord(chord)
```

(continues on next page)

(continued from previous page)

```
>>> abjad.f(chord)
<c' df'>4
```



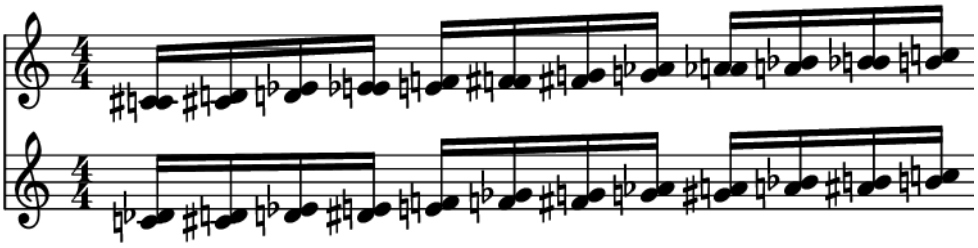
The example below shows the default spelling of 2-note chords by `Abjad` in the upper staff, and the respelt 2-note chords in the bottom staff.

```
>>> staff1 = abjad.Staff()
>>> staff2 = abjad.Staff()
>>> for pitch in range(12):
...     staff1.append(abjad.Chord([pitch, pitch + 1], (1, 16)))
...     chord = abjad.Chord([pitch, pitch + 1], (1, 16))
...     auxjad.respell_chord(chord)
...     staff2.append(chord)
>>> literal = abjad.LilyPondLiteral(r'\accidentalStyle dodecaphonic')
>>> abjad.attach(literal, staff1)
>>> abjad.attach(literal, staff2)
>>> score = abjad.Score([staff1, staff2])
>>> abjad.f(score)
\new Score
<<
    \new Staff
    {
        \accidentalStyle dodecaphonic
        <c' cs'>16
        <cs' d'>16
        <d' ef'>16
        <ef' e'>16
        <e' f'>16
        <f' fs'>16
        <fs' g'>16
        <g' af'>16
        <af' a'>16
        <a' bf'>16
        <bf' b'>16
        <b' c''>16
    }
    \new Staff
    {
        \accidentalStyle dodecaphonic
        <c' df'>16
        <cs' d'>16
        <d' ef'>16
        <ds' e'>16
        <e' f'>16
        <f' gf'>16
        <fs' g'>16
        <g' af'>16
        <gs' a'>16
        <a' bf'>16
        <as' b'>16
        <b' c''>16
```

(continues on next page)

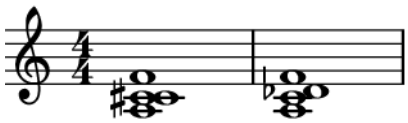
(continued from previous page)

```
    }
>>
```



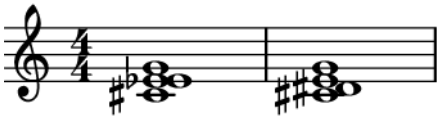
The function looks for all augmented unissons in chords of 3 or more pitches:

```
>>> chord1 = abjad.Chord(r"<a c' cs' f'>1")
>>> chord2 = abjad.Chord(r"<a c' cs' f'>1")
>>> auxjad.respell_chord(chord2)
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <a c' cs' f'>1
    <a c' df' f'>1
}
```



It is not a problem if the pitches are input out of order.

```
>>> chord1 = abjad.Chord(r"<e' cs' g' ef'>1")
>>> chord2 = abjad.Chord(r"<e' cs' g' ef'>1")
>>> auxjad.respell_chord(chord2)
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <cs' ef' e' g'>1
    <cs' ds' e' g'>1
}
```



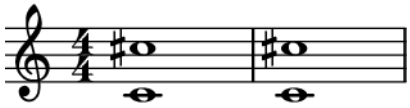
By default, this function only changes spelling for pitches that are 1 semitone apart.

```
>>> chord1 = abjad.Chord(r"<c' cs''>1")
>>> chord2 = abjad.Chord(r"<c' cs''>1")
>>> auxjad.respell_chord(chord2)
```

(continues on next page)

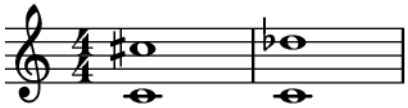
(continued from previous page)

```
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <c' cs''>1
    <c' cs''>1
}
```



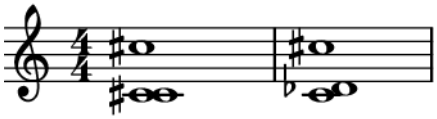
To consider pitches in different octaves (thus including augmented unisons, augmented octaves, augmented fifteenths, etc.), call this function with the keyword argument `include_multiples` set to `True`.

```
>>> chord1 = abjad.Chord(r"<c' cs''>1")
>>> chord2 = abjad.Chord(r"<c' cs''>1")
>>> auxjad.respell_chord(chord2, include_multiples=True)
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <c' cs''>1
    <c' df''>1
}
```



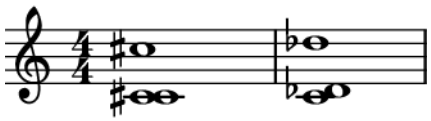
By default, when this function changes the spelling of a pitch, it does not change the spelling of all other pitches with the same pitch-class.

```
>>> chord1 = abjad.Chord(r"<c' cs' cs''>1")
>>> chord2 = abjad.Chord(r"<c' cs' cs''>1")
>>> auxjad.respell_chord(chord2)
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <c' cs' cs''>1
    <c' df' cs''>1
}
```



To alter all pitch-classes, call this function with the keyword argument `respell_by_pitch_class` set to `True`.

```
>>> chord1 = abjad.Chord(r"<c' cs' cs''>1")
>>> chord2 = abjad.Chord(r"<c' cs' cs''>1")
>>> auxjad.respell_chord(chord2, respell_by_pitch_class=True)
>>> staff = abjad.Staff([chord1, chord2])
>>> abjad.f(staff)
\new Staff
{
    <c' cs' cs''>1
    <c' df' df''>1
}
```



1.2.17 auxjad.rests_to_multimeasure_rest

auxjad.**rests_to_multimeasure_rest** (*container: abjad.core.Container.Container*)

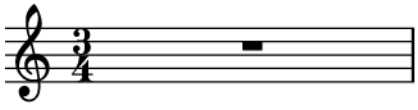
Mutates an input container (of type `abjad.Container` or child class) in place and has no return value. This function looks for bars filled with regular rests and converts them into an `abjad.MultimeasureRest`.

Converts any measure filled with regular rests into a measure with a single multi-measure rest.

```
>>> container = abjad.Container(r"r1")
>>> auxjad.rests_to_multimeasure_rest(container)
>>> abjad.f(container)
{
    R1
}
```

Works with measures with multiple regular rests.

```
>>> container = abjad.Container(r"\time 3/4 r4 r8.. r32 r4")
>>> auxjad.rests_to_multimeasure_rest(container)
>>> abjad.f(container)
{
    %%% \time 3/4 %%%
    R1 * 3/4
}
```

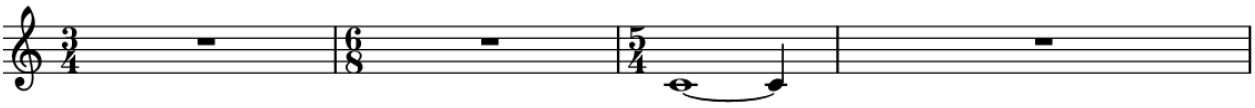


Note: Notice that the time signatures in the output are commented out with `%%%`. This is because Abjad only applies time signatures to containers that belong to a `abjad.Staff`. The present function works with either `abjad.Container` and `abjad.Staff`.

```
>>> container = abjad.Container(r"\time 3/4 r4 r8.. r32 r4")
>>> auxjad.rests_to_multimeasure_rest(container)
>>> abjad.f(container)
{
    %%% \time 3/4 %%%
    R1 * 3/4
}
>>> staff = abjad.Staff([container])
>>> abjad.f(container)
{
    %%% \time 3/4 %%%
    R1 * 3/4
}
```

Works with containers with multiple time signatures as well as notes.

```
>>> container = abjad.Container(r"\time 3/4 r2. | "
...                             "\time 6/8 r2. | "
...                             "\time 5/4 c'1 ~ c'4 | r1 r4"
...                             )
>>> auxjad.rests_to_multimeasure_rest(container)
>>> abjad.f(container)
{
    %%% \time 3/4 %%%
    R1 * 3/4
    %%% \time 6/8 %%%
    R1 * 3/4
    %%% \time 5/4 %%%
    c'1
    ~
    c'4
    R1 * 5/4
}
```



1.2.18 auxjad.Shuffler

class auxjad.**Shuffler**(*contents: abjad.core.Container.Container*, *, *output_single_measure: bool = False*, *disable_rewrite_meter: bool = False*, *force_time_signatures: bool = False*, *omit_time_signatures: bool = False*)

Shuffler takes an input `abjad.Container` and shuffles its logical ties. It can also shuffle only pitches, as well as rotate them. When shuffling or rotating pitches only, tuplets are allowed. Tuplets are not supported when shuffling leaves.

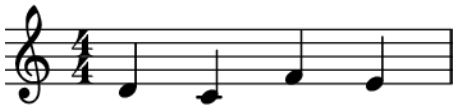
Calling the object will output a shuffled selection of the input container.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
```

(continues on next page)

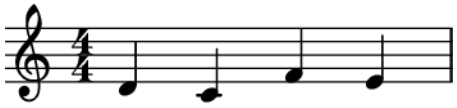
(continued from previous page)

```
\new Staff
{
    \time 4/4
    d'4
    c'4
    f'4
    e'4
}
```



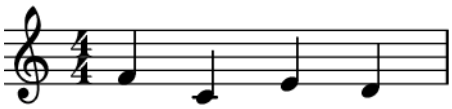
To get the result of the last operation, use the property `current_window`.

```
>>> music = shuffler.current_window
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    d'4
    c'4
    f'4
    e'4
}
```



Calling the object outputs the same result as using the method `shuffle()`.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.shuffle()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 4/4
    f'4
    c'4
    e'4
    d'4
}
```

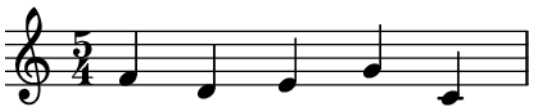


This class has many keyword arguments, all of which can be altered after instantiation using properties with the same names as shown below.

```
>>> container = abjad.Container(r"\time 3/4 c'4 d'4 e'4 |"
...                              r"\time 2/4 f'4 g'4 |"
...                              )
>>> shuffler = auxjad.Shuffler(container,
...                            output_single_measure=False,
...                            disable_rewrite_meter=False,
...                            force_time_signatures=False,
...                            omit_time_signatures=False,
...                            )
>>> shuffler.output_single_measure
False
>>> shuffler.disable_rewrite_meter
False
>>> shuffler.force_time_signatures
False
>>> shuffler.omit_time_signatures
False
>>> shuffler.output_single_measure = True
>>> shuffler.disable_rewrite_meter = True
>>> shuffler.force_time_signatures = True
>>> shuffler.omit_time_signatures = True
>>> shuffler.output_single_measure
True
>>> shuffler.disable_rewrite_meter
True
>>> shuffler.force_time_signatures
True
>>> shuffler.omit_time_signatures
True
```

If `output_single_measure` is set to `True`, then the whole container is output as a single measure, having its time signature rewritten.

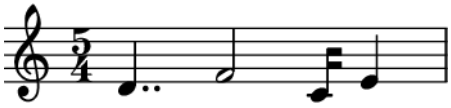
```
>>> container = abjad.Container(r"\time 3/4 c'4 d'4 e'4 |"
...                              r"\time 2/4 f'4 g'4"
...                              )
>>> shuffler = auxjad.Shuffler(container,
...                            output_single_measure=True,
...                            )
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 5/4
    f'4
    d'4
    e'4
    g'4
    c'4
}
```



If `disable_rewrite_meter` is set to `True`, then the automatic behaviour of rewriting the leaves according

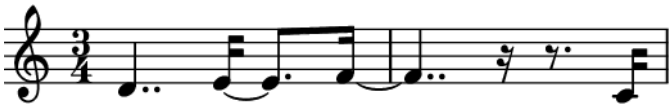
to the meter is disabled.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. e'4 |"
...                              r"\time 2/4 f'2"
...                              )
>>> shuffler = auxjad.Shuffler(container,
...                            output_single_measure=True,
...                            disable_rewrite_meter=True,
...                            )
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 5/4
    d'4..
    f'2
    c'16
    e'4
}
```



The first call to the instance will add the correct time signature to the first leaf. Subsequent calls will only add it if its necessary, such as when there is a time signature change in some bar in the container.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. e'4 | r4 f'2")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    d'4..
    e'16
    ~
    e'8.
    f'16
    ~
    f'4..
    r16
    r8.
    c'16
}
```

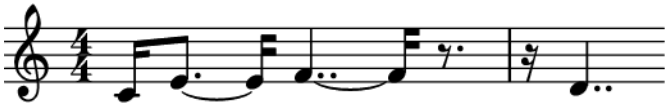


```
>>> music = shuffler()
>>> staff = abjad.Staff(music)
```

(continues on next page)

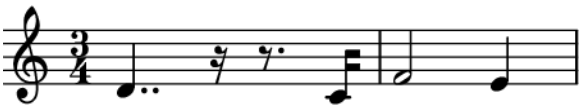
(continued from previous page)

```
>>> abjad.f(staff)
\new Staff
{
    c'16
    e'8.
    ~
    e'16
    f'4..
    ~
    f'16
    r8.
    r16
    d'4..
}
```



It is possible to force time signatures on every call using either optional keyword argument `force_time_signatures`.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. e'4 | r4 f'2")
>>> shuffler = auxjad.Shuffler(container,
...                            force_time_signatures=True,
...                            )
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    d'4..
    r16
    r8.
    c'16
    f'2
    e'4
}
```

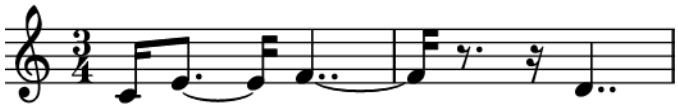


```
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    c'16
    e'8.
    ~
```

(continues on next page)

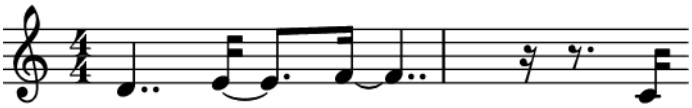
(continued from previous page)

```
    e'16
    f'4..
    ~
    f'16
    r8.
    r16
    d'4..
}
```



To disable time signatures altogether, initialise this class with the keyword argument `omit_time_signatures` set to `True` (default is `False`), or change the `omit_time_signatures` property after initialisation.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. e'4 | r4 f'2")
>>> shuffler = auxjad.Shuffler(container,
...                            omit_time_signatures=True,
...                            )
>>> music = shuffler()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    d'4..
    e'16
    ~
    e'8.
    f'16
    ~
    f'4..
    r16
    r8.
    c'16
}
```



```
>>> shuffler.omit_time_signatures
True
>>> shuffler.omit_time_signatures = False
>>> shuffler.omit_time_signatures
False
```

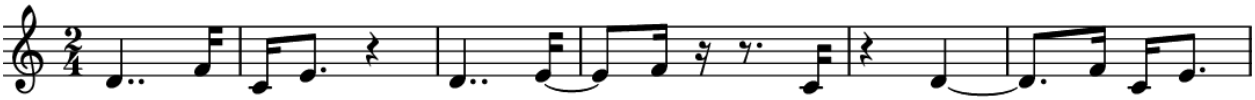
To output several shuffled containers at once, use the `output_n` method, inputting the desired number of iterations.

```
>>> container = abjad.Container(r"\time 2/4 c'16 d'4.. | r4 e'8. f'16")
>>> shuffler = auxjad.Shuffler(container)
```

(continues on next page)

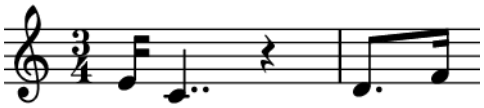
(continued from previous page)

```
>>> music = shuffler.output_n(3)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 2/4
    d'4..
    f'16
    c'16
    e'8.
    r4
    d'4..
    e'16
    ~
    e'8
    f'16
    r16
    r8.
    c'16
    r4
    d'4
    ~
    d'8.
    f'16
    c'16
    e'8.
}
```



To shuffle only pitches, keeping the durations of the leaves as they are, use the method `shuffle_pitches()`. It handles both notes and chords. Rests will remain at their current location.

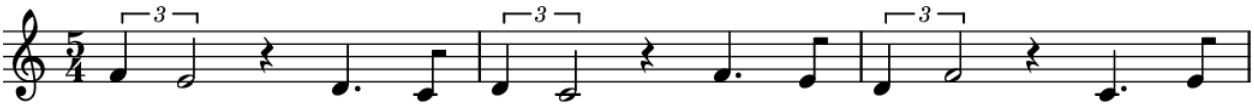
```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. | r4 e'8. f'16")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.shuffle_pitches()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    e'16
    c'4..
    r4
    d'8.
    f'16
}
```



When dealing with pitches, it is possible to use containers containing tuplets. And similarly to the method

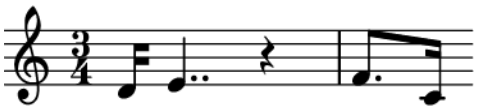
`output_n()`, to output several containers with shuffled pitches, use `output_n_shuffled_pitches()`.

```
>>> container = abjad.Container(r"\times 2/3 {\time 5/4 c'4 d'2}"
...                             r"r4 e'4. f'8"
...                             )
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.output_n_shuffled_pitches(3)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \times 2/3 {
        \time 5/4
        f'4
        e'2
    }
    r4
    d'4.
    c'8
    \times 2/3 {
        d'4
        c'2
    }
    r4
    f'4.
    e'8
    \times 2/3 {
        d'4
        f'2
    }
    r4
    c'4.
    e'8
}
```



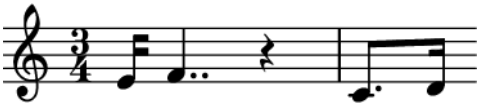
To rotate pitches, use the `rotate_pitches()` method.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. | r4 e'8. f'16")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.rotate_pitches()
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    d'16
    e'4..
    r4
    f'8.
    c'16
}
```



This method can take two optioanl keyword arguments: `anticlockwise`, set to `False` by default, and `n_rotations`, set to 1 by default. The first defines the direction of the rotation, while the later sets the number of rotations applied.

```
>>> container = abjad.Container(r"\time 3/4 c'16 d'4.. | r4 e'8. f'16")
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.rotate_pitches(anticlockwise=True, n_rotations=2)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \time 3/4
    e'16
    f'4..
    r4
    c'8.
    d'16
}
```



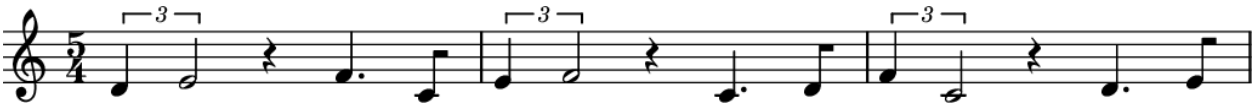
Similarly to the method `output_n()`, to output several containers with rotated pitches, use `output_n_rotated_pitches()`.

```
>>> container = abjad.Container(r"\times 2/3 {\time 5/4 c'4 d'2}"
...                             r"r4 e'4. f'8"
...                             )
>>> shuffler = auxjad.Shuffler(container)
>>> music = shuffler.output_n_rotated_pitches(3)
>>> staff = abjad.Staff(music)
>>> abjad.f(staff)
\new Staff
{
    \times 2/3 {
        \time 5/4
        d'4
        e'2
    }
    r4
    f'4.
    c'8
    \times 2/3 {
        e'4
        f'2
    }
    r4
    c'4.
    d'8
    \times 2/3 {
```

(continues on next page)

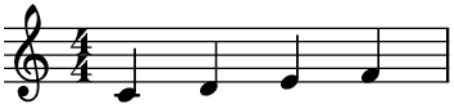
(continued from previous page)

```
        f'4
        c'2
    }
    r4
    d'4.
    e'8
}
```

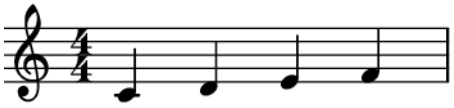


Use the property `contents` to get the input container upon which the shuffler operates. Notice that `contents` remains invariant after any shuffling or rotation operations (use `current_window` for the transformed selection of music). `contents` can be used to change the `abjad.Container` to be shuffled.

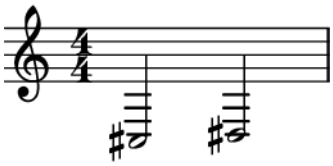
```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> shuffler = auxjad.Shuffler(container)
>>> abjad.f(shuffler.contents)
{
    c'4
    d'4
    e'4
    f'4
}
```



```
>>> shuffler()
>>> abjad.f(shuffler.contents)
{
    c'4
    d'4
    e'4
    f'4
}
```



```
>>> shuffler.contents = abjad.Container(r"cs2 ds2")
>>> abjad.f(shuffler.contents)
{
    cs2
    ds2
}
```



__init__(*contents: abjad.core.Container.Container*, *, *output_single_measure: bool = False*, *disable_rewrite_meter: bool = False*, *force_time_signatures: bool = False*, *omit_time_signatures: bool = False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, output_single_measure, ...)	Initialize self.
output_n(n)	Goes through n iterations of the shuffling process and outputs a single abjad.Selection.
output_n_rotated_pitches(n, *, n_rotations, ...)	Goes through n iterations of the pitch rotation process and outputs a single abjad.Selection.
output_n_shuffled_pitches(n)	Goes through n iterations of the pitch shuffling process and outputs a single abjad.Selection.
rotate_pitches(*, n_rotations, anticlockwise)	Rotates the pitches of contents.
shuffle()	Shuffles the logical ties of contents.
shuffle_pitches()	Shuffles only the pitches of contents.

Attributes

contents	The abjad.Container to be shuffled.
current_window	Read-only property, returns the result of the last operation.
disable_rewrite_meter	When True, the durations of the notes in the output will not be rewritten by the rewrite_meter mutation.
force_time_signatures	When True, every call will output a selection with a time signature.
omit_time_signatures	When True, the output will contain no time signatures.
output_single_measure	When True, the output will be a single measure even if the contents of the shuffler are several measures.

1.2.19 auxjad.simplified_time_signature_ratio

auxjad.**simplified_time_signature_ratio**(*ratio: (<class 'tuple'>, <class 'abjad.indicators.TimeSignature.TimeSignature'>, <class 'abjad.utilities.Duration.Duration'>, <class 'abjad.meter.Meter'>)*, ***, *min_denominator: int = 4*, *output_pair_of_int: bool = False*) -> (<class 'abjad.indicators.TimeSignature.TimeSignature'>, <class 'tuple'>)

Returns an `abjad.TimeSignature` with the simplified ratio of an input ratio according to a minimum denominator value. The input ratio can be a `tuple` of integers, an `abjad.TimeSignature`, `abjad.Duration`, or an `abjad.Meter`).

By default, the function simplifies the ratio of numerator/denominator using a minimum denominator value of 4 (that is, the denominator will not get smaller than 4). In the case below, (2, 4) is the simplest representation of the ratio (4, 8) with a denominator equal to or larger than 4.

```
>>> time_signature = auxjad.simplified_time_signature_ratio((4, 8))
>>> format(time_signature)
abjad.TimeSignature((2, 4))
>>> time_signature = auxjad.simplified_time_signature_ratio((1, 1))
>>> format(time_signature)
abjad.TimeSignature((4, 4))
```

If a ratio cannot be simplified at all, the function returns a time signature with the original ratio.

```
>>> time_signature = auxjad.simplified_time_signature_ratio((7, 8))
>>> format(time_signature)
abjad.TimeSignature((7, 8))
```

The `min_denominator` can be set to values other than 4. If set to 2, the simplest representaion of the ratio (4, 8) becomes (1, 2).

```
>>> time_signature = auxjad.simplified_time_signature_ratio(
...     (4, 8),
...     min_denominator=2,
... )
>>> format(time_signature)
abjad.TimeSignature((1, 2))
>>> time_signature = auxjad.simplified_time_signature_ratio(
...     (1, 1),
...     min_denominator=1,
... )
>>> format(time_signature)
abjad.TimeSignature((1, 1))
```

By default, the function returns an `abjad.TimeSignature` for whatever type of argument it receives (which can be a `tuple` of integers, an `abjad.TimeSignature`, an `abjad.Duration`, or an `abjad.Meter`).

```
>>> arg = (4, 8)
>>> time_signature = auxjad.simplified_time_signature_ratio(arg)
>>> format(time_signature)
abjad.TimeSignature((2, 4))
```

```
>>> arg = abjad.Duration((4, 8))
>>> time_signature = auxjad.simplified_time_signature_ratio(arg)
```

(continues on next page)

(continued from previous page)

```
>>> format(time_signature)
abjad.TimeSignature((2, 4))
```

```
>>> arg = abjad.Meter((4, 8))
>>> time_signature = auxjad.simplified_time_signature_ratio(arg)
>>> format(time_signature)
abjad.TimeSignature((2, 4))
```

```
>>> arg = abjad.TimeSignature((4, 8))
>>> time_signature = auxjad.simplified_time_signature_ratio(arg)
>>> format(time_signature)
abjad.TimeSignature((2, 4))
```

Call the function with the keyword argument `output_pair_of_int` set to `True` and the output will be a `tuple` of integers, regardless of the input argument.

```
>>> arg = (4, 8)
>>> pair = auxjad.simplified_time_signature_ratio(
...     arg,
...     output_pair_of_int=True,
... )
>>> pair
(2, 4)
```

```
>>> arg = abjad.Duration((4, 8))
>>> pair = auxjad.simplified_time_signature_ratio(
...     arg,
...     output_pair_of_int=True,
... )
>>> assert pair == (2, 4)
```

arg = abjad.Meter((4, 8)) >>> pair = auxjad.simplified_time_signature_ratio(... arg, ... output_pair_of_int=True, ...) >>> pair (2, 4)

arg = abjad.TimeSignature((4, 8)) >>> pair = auxjad.simplified_time_signature_ratio(... arg, ... output_pair_of_int=True, ...) >>> pair (2, 4)

1.2.20 auxjad.sync_containers

auxjad.**sync_containers**(**containers*, *use_multimeasure_rests: bool = True*, *adjust_last_time_signature: bool = True*)

Mutates two or more input containers (of type `abjad.Container` or child class) in place and has no return value. This function finds the longest container among the inputs and adds rests to all the shorter ones, making them the same length. By default, it rewrites the last time signature if necessary, and uses multi-measure rests whenever possible.

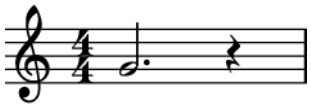
Input two or more containers. This function will fill the shortest ones with rests ensuring all their lengths become the same.

```
>>> container1 = abjad.Container(r"\time 4/4 g'2.")
>>> container2 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1, container2)
>>> abjad.f(container1)
{
```

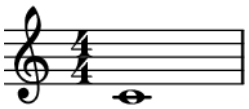
(continues on next page)

(continued from previous page)

```
    %%% \time 4/4 %%%
    g'2.
    r4
}
```



```
>>> abjad.f(container2)
{
    %%% \time 4/4 %%%
    c'1
}
```

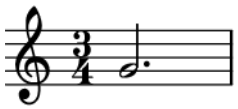


Note: Notice that the time signatures in the output are commented out with `%%%`. This is because Abjad only applies time signatures to containers that belong to a `abjad.Staff`. The present function works with either `abjad.Container` and `abjad.Staff`.

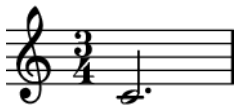
```
>>> container1 = abjad.Container(r"\time 4/4 g'2.")
>>> container2 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1, container2)
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    g'2.
    r4
}
>>> staff = abjad.Staff([container1])
>>> abjad.f(container1)
{
    \time 4/4
    g'2.
    r4
}
```

If all containers have the same size, no modification is applied.

```
>>> container1 = abjad.Container(r"\time 3/4 g'2.")
>>> container2 = abjad.Container(r"\time 3/4 c'2.")
>>> auxjad.sync_containers(container1, container2)
>>> abjad.f(container1)
{
    %%% \time 3/4 %%%
    g'2.
}
```

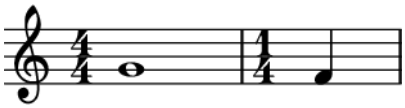


```
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    c'2.
}
```

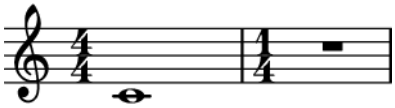


By default, this function closes the longest container by rewriting the time signature of its last bar if necessary (if it is underfull), and uses multi-measure rests whenever possible.

```
>>> container1 = abjad.Container(r"\time 4/4 g'1 | f'4")
>>> container2 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1, container2)
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    g'1
    %%% \time 1/4 %%%
    f'4
}
```



```
>>> abjad.f(container2)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 1/4 %%%
    R1*1/4
}
```



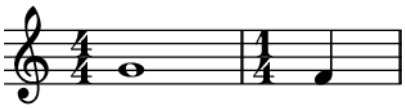
To disable multi-measure rests, set the keyword argument `use_multimeasure_rests` to `False`.

```
>>> container1 = abjad.Container(r"\time 4/4 g'1 | f'4")
>>> container2 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1,
...                        container2,
...                        use_multimeasure_rests=False,
```

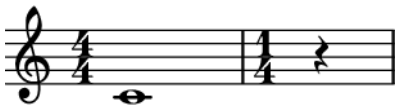
(continues on next page)

(continued from previous page)

```
...                         )
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    g'1
    %%% \time 1/4 %%%
    f'4
}
```

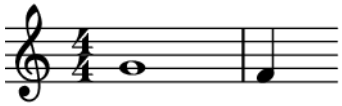


```
>>> abjad.f(container2)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 1/4 %%%
    r4
}
```

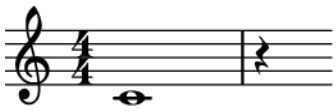


To allow containers to be left open (with underfull bars), set the keyword argument `adjust_last_time_signature` to `False`.

```
>>> container1 = abjad.Container(r"\time 4/4 g'1 | f'4")
>>> container2 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1,
...                        container2,
...                        adjust_last_time_signature=False,
...                        )
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    g'1
    f'4
}
```

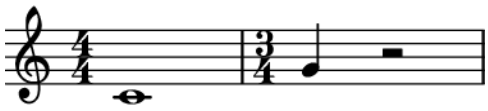


```
>>> abjad.f(container2)
{
    %%% \time 4/4 %%%
    c'1
    r4
}
```

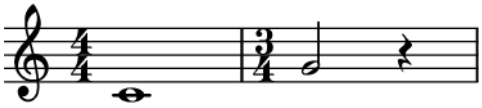


This function can take an arbitrary number of containers.

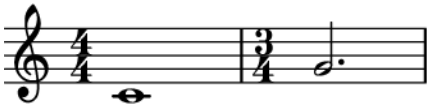
```
>>> container1 = abjad.Container(r"\time 4/4 c'1 | g'4")
>>> container2 = abjad.Container(r"\time 4/4 c'1 | g'2")
>>> container3 = abjad.Container(r"\time 4/4 c'1 | g'2.")
>>> container4 = abjad.Container(r"\time 4/4 c'1")
>>> auxjad.sync_containers(container1,
...                        container2,
...                        container3,
...                        container4,
...                        )
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 3/4 %%%
    g'4
    r2
}
```



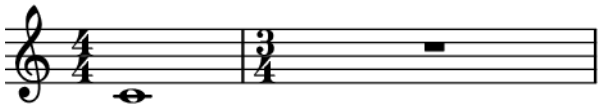
```
>>> abjad.f(container2)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 3/4 %%%
    g'2
    r4
}
```



```
>>> abjad.f(container3)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 3/4 %%%
    g'2.
}
```

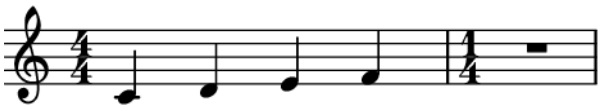


```
>>> abjad.f(container4)
{
    %%% \time 4/4 %%%
    c'1
    %%% \time 3/4 %%%
    R1*3/4
}
```

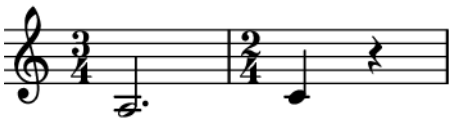


The containers can be of different length, can have different time signatures, and can contain time signature changes as well.

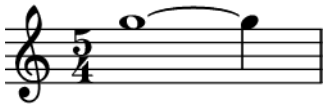
```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 4/4 c'4")
>>> container3 = abjad.Container(r"\time 5/4 g''1 ~ g''4")
>>> container4 = abjad.Container(r"\time 6/8 c'2")
>>> auxjad.sync_containers(container1,
...                        container2,
...                        container3,
...                        container4,
...                        )
>>> abjad.f(container1)
{
    %%% \time 4/4 %%%
    c'4
    d'4
    e'4
    f'4
    %%% \time 1/4 %%%
    R1*1/4
}
```



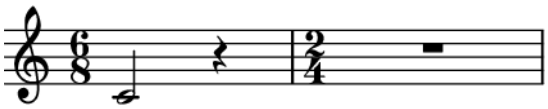
```
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    a2.
    %%% \time 2/4 %%%
    c'4
    r4
}
```



```
>>> abjad.f(container3)
{
    %%% \time 5/4 %%%
    g''1
    ~
    g''4
}
```



```
>>> abjad.f(container4)
{
    %%% \time 6/8 %%%
    c'2
    r4
    %%% \time 2/4 %%%
    R1*1/2
}
```



It's important to note that LilyPond does not support simultanoues staves with different time signatures (i.e. polymetric notation) by default. In order to enable it, the `"Timing_translator"` and `"Default_bar_line_engraver"` must be removed from the `Score` context and added to the `Staff` context. Below is a full example of how this can be accomplished using Abjad.

```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 4/4 c'4")
>>> container3 = abjad.Container(r"\time 5/4 g''1 ~ g''4")
>>> container4 = abjad.Container(r"\time 6/8 c'2")
>>> auxjad.sync_containers(container1,
...                        container2,
...                        container3,
...                        container4,
...                        )
>>> staves = [abjad.Staff([container1]),
...           abjad.Staff([container2]),
...           abjad.Staff([container3]),
...           abjad.Staff([container4]),
...           ]
>>> score = abjad.Score(staves)
>>> lilypond_file = abjad.LilyPondFile.new()
>>> score_block = abjad.Block(name='score')
>>> layout_block = abjad.Block(name='layout')
>>> score_block.items.append(score)
>>> score_block.items.append(layout_block)
>>> lilypond_file.items.append(score_block)
>>> layout_block.items.append(
...     r'''
```

(continues on next page)

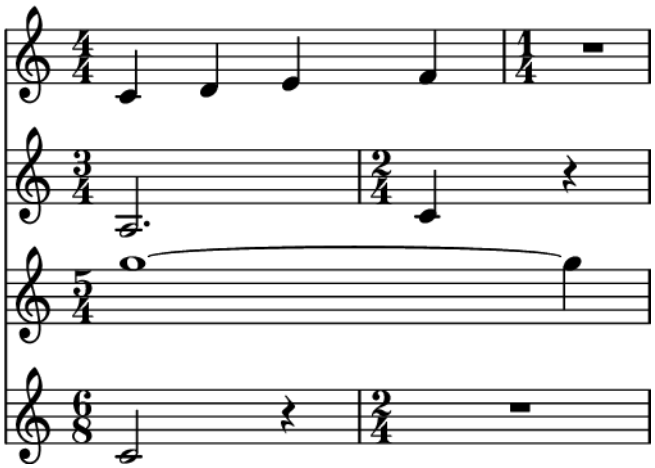
(continued from previous page)

```
...     \context {
...         \Score
...         \remove "Timing_translator"
...         \remove "Default_bar_line_engraver"
...     }
...     \context {
...         \Staff
...         \consists "Timing_translator"
...         \consists "Default_bar_line_engraver"
...     }
...     ''')
>>> abjad.f(lilypond_file)
\score { %! abjad.LilyPondFile._get_formatted_blocks()
    \new Score
    <<
        \new Staff
        {
            {
                \time 4/4
                c'4
                d'4
                e'4
                f'4
                \time 1/4
                R1 * 1/4
            }
        }
        \new Staff
        {
            {
                \time 3/4
                a2.
                \time 2/4
                c'4
                r4
            }
        }
        \new Staff
        {
            {
                \time 5/4
                g''1
                ~
                g''4
            }
        }
        \new Staff
        {
            {
                \time 6/8
                c'2
                r4
                \time 2/4
                R1 * 1/2
            }
        }
    >>
```

(continues on next page)

(continued from previous page)

```
    \layout {
        \context {
            \Score
            \remove "Timing_translator"
            \remove "Default_bar_line_engraver"
        }
        \context {
            \Staff
            \consists "Timing_translator"
            \consists "Default_bar_line_engraver"
        }
    }
} %! abjad.LilyPondFile._get_formatted_blocks()
```



Warning: If one or more containers is malformed, i.e. it has an underfilled bar before a time signature change, the function raises a `ValueError` exception.

```
>>> container1 = abjad.Container(r"\time 4/4 g'1 | f'4")
>>> container2 = abjad.Container(r"\time 5/4 c'1 | \time 4/4 d'4")
>>> auxjad.sync_containers(container1, container2)
ValueError: at least one 'container' is malformed, with an underfull
bar preceeding a time signature change
```

1.2.21 auxjad.TenneySelector

class auxjad.**TenneySelector**(*contents: list*, *, *weights: list = None*, *curvature: float = 1.0*)

This in an implementation of the Dissonant Counterpoint Algorithm by James Tenney. This class can be used to randomly select elements from an input list, giving more weight to elements which have not been selected in recent iterations. In other words, Tenney's algorithm uses feedback in order to lower the weight of recently selected elements.

This implementation is based on the paper: Polansky, L., A. Barnett, and M. Winter (2011). 'A Few More Words About James Tenney: Dissonant Counterpoint and Statistical Feedback'. In: Journal of Mathematics and Music 5(2). pp. 63–82.

The selector should be initialised with a list of objects. The contents of the list can be absolutely anything.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> selector.contents
['A', 'B', 'C', 'D', 'E', 'F']
```

Applying the `len()` function to the selector will give the length of the input list.

```
>>> len(selector)
6
```

When no other keyword arguments are used, the default probabilities of each element in the list is 1.0. Probabilities are not normalised. Use the previous_index attribute to check the previously selected index (default is None).

```
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.previous_index
None
```

Calling the selector will output one of its elements, selected according to the current probability values.

```
>>> selector()
C
```

Alternatively, use the `next()` function or `__next__()` method to get the next result.

```
>>> selector.__next__()
A
>>> next(selector)
D
```

After each call, the object updates all probability values, setting the previously selected element's probability at 0.0 and raising all other probabilities according to a growth function (more on this below).

```
>>> result = ''
>>> for _ in range(30):
...     result += selector()
>>> result
EDFACEABAFDCEDAFADCBFEDABEDFEC
```

From the result above it is possible to see that there are no immediate repetitions of elements (since once selected, their probability is always set to 0.0 and will take at least one iteration to grow to a non-zero value). Checking the probabilities and previous_index attributes will give us their current values.

```
>>> selector.probabilities
[6.0, 5.0, 0.0, 3.0, 1.0, 2.0]
>>> selector.previous_index
2
```

This class can take two optional keywords argument during its instantiation, namely `weights` and `curvature`. `weights` takes a list of floats with the individual weights of each element; by default, all weights are set to 1.0. These weights affects the effective probability of each element. The other argument, `curvature`, is the exponent of the growth function for all elements. The growth function takes as input the number of iterations since an element has been last selected, and raise this number by the curvature value. If `curvature` is set to 1.0 (which is its default value), the growth is linear with each iteration. If set to a value larger than 0.0 and less than 1.0, the growth is negative (or concave), so that the chances of an element which is not being selected will grow at ever smaller rates as the number of iterations it has not been selected increase. If the `curvature` is set to 1.0, the growth is linear with the number of iterations. If the `curvature` is larger

than 1.0, the curvature is positive (or convex) and the growth will accelerate as the number of iterations an element has not been selected grows. Setting the curvature to 0.0 will result in an static probability vector with all values set to 1.0, except for the previously selected one which will be set to 0.0; this will result in a uniformly random selection without repetition.

With linear curvature (default value of 1.0):

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> selector.curvature
1.0
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector()
'B'
>>> selector.curvature
1.0
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[2.0, 0.0, 2.0, 2.0, 2.0, 2.0]
```

Using a convex curvature:

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'],
...                                  curvature=0.2,
...                                  )
>>> selector.curvature
0.2
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector()
'C'
>>> selector.curvature
0.2
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[1.148698354997035, 1.148698354997035, 0.0, 1.148698354997035,
1.148698354997035, 1.148698354997035]
```

With a convex curvature, the growth of the probability of each non-selected term gets smaller as the number of times it is not selected increases. The smaller the curvature is, the less difference there will be between any non-previously selected elements. This results in sequences which have more chances of a same element being near each other. In the sequence below, note how there are many cases of a same element being separated only by a single other one, such as `'ACA'` in index 6.

```
>>> result = ''
>>> for _ in range(30):
...     result += selector()
>>> result
DACBEDFACABDACECBEFAEDBAFBABFD
```

Checking the probability values at this point outputs:

```
>>> selector.probabilities
[1.2457309396155174, 1.148698354997035, 1.6952182030724354, 0.0,
1.5518455739153598, 1.0]
```

As we can see, all non-zero values are relatively close to each other, which is why there is a high chance of an element being selected again just two iterations apart.

Using a concave curvature:

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'],
...                                  curvature=15.2,
...                                  )
>>> selector.curvature
0.2
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector()
'C'
>>> selector.curvature
0.2
>>> selector.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[37640.547696542824, 37640.547696542824, 37640.547696542824, 0.0,
37640.547696542824, 37640.547696542824]
```

With a concave curvature, the growth of the probability of each non-selected term gets larger as the number of times it is not selected increases. The larger the curvature is, the larger difference there will be between any non-previously selected elements. This results in sequences which have less chances of a same element being near each other. In the sequence below, with a curvature of 15.2, note how the elements are as far apart from each other, resulting in a repeating string of `'DFAECB'`.

```
>>> result = ''
>>> for _ in range(30):
...     result += selector()
>>> result
DFAECBDFAECBDFAECBDFAECBDFAECB
```

Checking the probability values at this point outputs:

```
>>> selector.probabilities
[17874877.39956566, 0.0, 1.0, 42106007735.02238,
37640.547696542824, 1416810830.8957152]
```

As we can see, the non-zero values vary wildly. The higher the curvature, the higher the difference between these values, making some of them much more likely to be selected.

Each element can also have a fixed weight to themselves. This will affect the probability calculation. The example below uses the default linear curvature.

```
>>> selector = auxjad.TenneySelector(
...     ['A', 'B', 'C', 'D', 'E', 'F'],
...     weights=[1.0, 1.0, 5.0, 5.0, 10.0, 20.0],
>>> )
>>> selector.weights
```

(continues on next page)

(continued from previous page)

```
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> selector.probabilities
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> result = ''
>>> for _ in range(30):
...     result += selector()
>>> result
FBEFECFDEADFEDFEDBFECDAFCEDCFE
>>> selector.weights
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> selector.probabilities
[7.0, 12.0, 10.0, 15.0, 0.0, 20.0]
```

To reset the probability distribution of all elements to its initial value (an uniform distribution), use the method `reset_probabilities()`.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> for _ in range(30):
...     selector()
>>> selector.probabilities
[4.0, 3.0, 1.0, 0.0, 5.0, 2.0]
>>> selector.reset_probabilities()
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

This class allows slicing to get and set values of contents of the selector. This will not affect the current probability vector, and the new element will have the same probability as the one it replaced.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> for _ in range(30):
...     selector()
>>> selector.probabilities
[3.0, 2.0, 1.0, 7.0, 5.0, 0.0]
>>> selector[2]
'C'
>>> selector[1:4]
['B', 'C', 'D']
>>> selector[2] = 'foo'
>>> selector.contents
['A', 'B', 'foo', 'D', 'E', 'F']
>>> selector[:] = ['foo', 'bar', 'X', 'Y', 'Z', '...']
>>> selector.contents
['foo', 'bar', 'X', 'Y', 'Z', '...']
>>> selector.probabilities
[3.0, 2.0, 1.0, 7.0, 5.0, 0.0]
```

You can also check if the instance contains a specific element. In the case of the selector above, we have:

```
>>> 'foo' in selector
True
>>> 'A' in selector
False
```

A new list of an arbitrary length can be set at any point using the property `contents`. Do notice that the probabilities will be reset at that point. This method can take the optional keyword argument weights similarly to when instantiating the class.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> for _ in range(30):
...     selector()
>>> selector.probabilities
[2.0, 1.0, 4.0, 3.0, 0.0, 5.0]
>>> selector.contents
['A', 'B', 'C', 'D', 'E', 'F']
>>> selector.contents = [2, 4, 6, 8]
>>> selector.contents
[2, 4, 6, 8]
>>> len(selector)
4
>>> selector.weights
[1.0, 1.0, 1.0, 1.0]
>>> selector.probabilities
[1.0, 1.0, 1.0, 1.0]
```

To change the curvature value at any point, simply set the property `curvature` to a different value.

```
>>> selector = auxjad.TenneySelector(['A', 'B', 'C', 'D', 'E', 'F'])
>>> selector.curvature
1.0
>>> selector.curvature = 0.25
>>> selector.curvature
0.25
```

__init__(*contents: list*, ***, *weights: list = None*, *curvature: float = 1.0*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(contents, *, weights, curvature)	Initialize self.
reset_probabilities()	Resets the probability distribution of all elements to an uniform distribution.

Attributes

contents	The list from which the selector picks elements.
curvature	The exponent of the growth function.
previous_index	Read-only property, returns the index of the previously output element.
weights	The list with weights for each element of contents.

1.2.22 auxjad.underfull_duration

auxjad.**underfull_duration**(*container: abjad.core.Container.Container*) → abjad.utilities.Duration.Duration
Returns the missing abjad.Duration of an underfull container (of type abjad.Container or child class).

Returns the missing duration of the last bar of any container or child class. If no time signature is encountered, it uses LilyPond's convention and considers the container as in 4/4.

```
>>> container1 = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"c'4 d'4 e'4")
>>> container3 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4")
>>> container4 = abjad.Container(r"c'4 d'4 e'4 f'4 | c'4 d'4 e'4 f'4")
>>> auxjad.underfull_duration(container1)
0
>>> auxjad.underfull_duration(container2)
1/4
>>> auxjad.underfull_duration(container3)
3/4
>>> auxjad.underfull_duration(container4)
0
```

Handles any time signatures as well as changes of time signature.

```
>>> container1 = abjad.Container(r"\time 4/4 c'4 d'4 e'4 f'4")
>>> container2 = abjad.Container(r"\time 3/4 a2. \time 2/4 r2")
>>> container3 = abjad.Container(r"\time 5/4 g1 ~ g4 \time 4/4 af'2")
>>> container4 = abjad.Container(r"\time 6/8 c'2 ~ c'8")
>>> auxjad.underfull_duration(container1)
0
>>> auxjad.underfull_duration(container2)
0
>>> auxjad.underfull_duration(container3)
1/2
>>> auxjad.underfull_duration(container4)
1/8
```

Correctly handles partial time signatures.

```
>>> container = abjad.Container(r"c'4 d'4 e'4 f'4")
>>> time_signature = abjad.TimeSignature((3, 4), partial=(1, 4))
>>> abjad.attach(time_signature, container[0])
>>> auxjad.underfull_duration(container)
0
```

It also handles multi-measure rests.

```
>>> container1 = abjad.Container(r"R1")
>>> container2 = abjad.Container(r"\time 3/4 R1*3/4 \time 2/4 r2")
>>> container3 = abjad.Container(r"\time 5/4 R1*5/4 \time 4/4 g''4")
>>> container4 = abjad.Container(r"\time 6/8 R1*1/2")
>>> auxjad.underfull_duration(container1)
0
>>> auxjad.underfull_duration(container2)
0
>>> auxjad.underfull_duration(container3)
3/4
>>> auxjad.underfull_duration(container4)
1/4
```

Warning: If a container is malformed, i.e. it has an underfilled bar before a time signature change, the function raises a `ValueError` exception.

```
>>> container = abjad.Container(r"\time 5/4 g''1 \time 4/4 f'1")
>>> auxjad.underfull_duration(container)
ValueError: 'container' is malformed, with an underfull bar preceeding
a time signature change
```

1.3 Indices and tables

- genindex
- modindex
- search

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

C

F

H

L

R

S

T

U