auxjad Documentation

Release 0.4.0

Gilberto Agostinho

Apr 12, 2020

CONTENTS

CHAPTER

ONE

THE AUXJAD PACKAGE

1.1 auxjad

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.

Bugs can be reported to https://github.com/gilbertohasnofb/auxjad/issues.

This library is published under the MIT License.

1.2 are_containers_equal

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

A comparator function returning `True` when two containers are identical and `False` when they are 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.are_containers_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.are_containers_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")
```

(continues on next page)

(continued from previous page)

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

1.3 are_leaves_tieable

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

A comparator function returning `True` when two leaves have identical pitches and thus can be tied, otherwise returning `False`.

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.are_leaves_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.are_leaves_tieable(Leaf1, Leaf2)
True
>>> auxjad.are_leaves_tieable(Leaf1, Leaf3)
False
>>> auxjad.are_leaves_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.are_leaves_tieable(chord1, chord2)
True
>>> auxjad.are_leaves_tieable(chord1, chord3)
False
>>> auxjad.are_leaves_tieable(chord2, chord3)
False
```

Leaves can also be part of containers.

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

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

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

1.4 CartographyContainer

class `auxjad.`**`CartographyContainer`**(*container: list*, *, *decay_rate: float = 0.75*)

A container used to store, manipulate, and select objects using a decaying weighted function.

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

```
>>> container = auxjad.CartographyContainer([0, 1, 2, 3, 4])
>>> container.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.

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

Applying the `len()` function to the container will give the length of the container.

```
>>> len(container)
5
```

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

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

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

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

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

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

The decay rate can also be set after the creation of a container, using the method `set_decay_rate()`.

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

Appending is a type of container transformation. It discards the first element of the container, shifts all others leftwards, and then appends the new element to the rightmost index.

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

The method `append_keeping_n()` is similar to `append()`, but it keeps the first n indeces untouched. It thus discards the n+1-th element, shifts all the next ones lefwards and then appends the new element at the end of the container.

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

Prepending is another type of container transformation. It discards the last element of the container, shifts all others rightwards, and then prepends the new element to the leftmost index.

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

Rotation is another type of container transformation. It rotates all elements rightwards, while moving the rightmost element into the leftmost index. It can take the optional keyword argument anticlockwise which if set to `True` will rotate in the opposite direction.

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

It is also possible to mirror two elements around a pivot at the centre of the container; given an element (selected by its index), this operation will locate and swap it for its complementary element. The complementary element is defined as that one which is at a same distance from the centre pivot but in the opposite direction.

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

To mirror a random pair of complementary elements, use the `mirror_random()` method. In case of a container 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.

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

The method `randomise()` will randomise the position of the elements of a container.

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

The contents of a container can also be altered after it has been initialised using the `set_container()` method. The length of the container can change too.

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

To method `replace_element()` replaces a specific element at a specified index.

```
>>> container = auxjad.CartographyContainer([10, 7, 14, 31, 98])
>>> container.contents
[10, 7, 14, 31, 98]
>>> container.replace_element(100, 2)
>>> container.contents
[10, 7, 100, 31, 98]
```

The attribute previous_index stores the previously selected index. It can be used with the `get_element()` method in order to retrieve the last value output by the object.

```
>>> container = auxjad.CartographyContainer([10, 7, 14, 31, 98])
>>> container()
31
```

(continues on next page)

(continued from previous page)

```
>>> previous_index = container.previous_index
>>> previous_index
3
>>> container.get_element(previous_index)
31
```

Individual elements are also accessible via indeces of the content attribute. When accessed in this manner, they can also be sliced like a regular list.

```
>>> container = auxjad.CartographyContainer([10, 7, 14, 31, 98])
>>> container.contents[2]
14
>>> container.contents[1:4]
[7, 14, 31]
```

1.5 close_container

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

Changes the time signature of the last bar of an underfull `abjad.Container` in order to make it full, if necessary.

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

(continues on next page)

(continued from previous page)

```
}
>>> 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.
    %%% \time 1/4 %%%
    c'4
}
>>> abjad.f(container3)
{
    %%% \time 5/4 %%%
    g1
    ~
    g4
    %%% \time 2/4 %%%
    af'2
}
```

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)
{
```

(continues on next page)

(continued from previous page)

```
    %%% \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
}
```

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
```

1.6 fill_with_rests

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

Fills an `abjad.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
    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
```

(continues on next page)

(continued from previous page)

```
    f'4
    g'4
    r2.
}
>>> abjad.f(container2)
{
    %%% \time 3/4 %%%
    a2.
    %%% \time 2/4 %%%
    c'4
    r4
}
>>> abjad.f(container3)
{
    %%% \time 5/4 %%%
    g1
    ~
    g4
    %%% \time 4/4 %%%
    af'2
    r2
}
```

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])
```

(continues on next page)

(continued from previous page)

```
>>> auxjad.fill_with_rests(container)
>>> abjad.f(container)
{
    %%% \partial 4 %%%
    %%% \time 3/4 %%%
    c'4
    d'4
    e'4
    f'4
    g'4
    r2
}
```

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
```

1.7 is_container_full

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

Checks if an `abjad.Container` is full. Based on `auxjad.underfull_duration`.

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.is_container_full(container1)
True
>>> auxjad.is_container_full(container2)
False
>>> auxjad.is_container_full(container3)
False
>>> auxjad.is_container_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.is_container_full(container1)
True
>>> auxjad.is_container_full(container2)
True
>>> auxjad.is_container_full(container3)
False
```

(continues on next page)

(continued from previous page)

```
>>> auxjad.is_container_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.is_container_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.is_container_full(container1)
True
>>> auxjad.is_container_full(container2)
True
>>> auxjad.is_container_full(container3)
False
>>> auxjad.is_container_full(container4)
False
```

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.is_container_full(container)
ValueError: 'container' is malformed, with an underfull bar preceeding
a time signature change
```

1.8 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*)

An extension of `abjad.LeafMaker` which can also 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
```

(continues on next page)

(continued from previous page)

```
{
    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 `no_repeat`:

```
>>> 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,
...                        no_repeat=True,
...                        )
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
```

(continues on next page)

(continued from previous page)

```
\new Staff
{
    c'32
    \pp
    d'16
    e'16.
    \mp
    f'8
    \f
    g'8
    ~
    g'32
    a'8.
    \p
}
```

The lengths of both `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 length of `articulations` is 1, it will apply to all elements. If the length of `dynamics` is 1, it will apply to the first element only:

```
>>> 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
```

(continues on next page)

(continued from previous page)

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

1.9 LoopWindow

class auxjad.**LoopWindow**(*container: 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*, *head_position: (<class 'int'>, <class 'float'>, <class 'tuple'>, <class 'str'>, <class 'abjad.utilities.Duration.Duration'>) = 0*, *omit_time_signature: bool = False*)

Using a looping window, this slices an input `abjad.Container` and output them as containers.

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.LoopWindow(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
}
```

The method `get_current_window()` will output the current window without moving the head forwards.

```
>>> notes = looper.get_current_window()
>>> 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 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.LoopWindow(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 `LoopWindow` 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.LoopWindow(input_music,
...                            window_size=(3, 4),
...                            step_size=(1, 8),
...                            )
>>> for window in looper:
...     staff = abjad.Staff(window)
...     abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'2
}
\new Staff
{
    c'8
    d'8
    ~
    d'4.
    e'8
}
```

(continues on next page)

(continued from previous page)

```
\new Staff
{
    d'2
    e'4
}
\new Staff
{
    d'4.
    e'8
    ~
    e'8
    r8
}
\new Staff
{
    d'4
    e'4
    r4
}
\new Staff
{
    d'8
    e'8
    ~
    e'8
    r4.
}
\new Staff
{
    e'4
    r2
}
\new Staff
{
    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). 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.LoopWindow(input_music,
...                            window_size=(3, 4),
...                            step_size=(5, 8),
...                            max_steps=2,
...                            repetition_chance=0.25,
...                            head_position=(2, 8),
...                            omit_time_signature=False,
...                            )
>>> looper.window_size
```

(continues on next page)

(continued from previous page)

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

Use the `set_` methods below to change these values after initialisation.

```
>>> looper.set_window_size((5, 4))
>>> looper.set_step_size((1, 4))
>>> looper.set_max_steps(3)
>>> looper.set_repetition_chance(0.1)
>>> looper.set_head_position(0)
>>> looper.set_omit_time_signature(True)
>>> looper.window_size
5/4
>>> looper.step_size
1/4
>>> looper.max_steps
3
>>> looper.repetition_chance
0.1
>>> looper.head_position
0
>>> looper.omit_time_signature
True
```

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.LoopWindow(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
```

(continues on next page)

(continued from previous page)

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

To change the size of the looping window after instantiation, use the method `set_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.LoopWindow(input_music)
>>> for _ in range(3):
...     notes = looper()
...     staff = abjad.Staff(notes)
...     abjad.f(staff)
\new Staff
{
    \time 4/4
    c'4
    d'2
    e'4
}
\new Staff
```

(continues on next page)

(continued from previous page)

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

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

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

(continues on next page)

(continued from previous page)

```
    e'4
}
```

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

1.10 LoopWindowByElements

class auxjad.**LoopWindowByElements**(*container: abjad.core.Container.Container*, *, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *head_position: int = 0*, *omit_time_signature: bool = False*)

Takes an `abjad.Container` as input as well as an integer representing the number of elements per looping window, then outputs a container with the elements processed in the looping process. For instance, if the initial container had the leaves `[A, B, C, D, E, F]` and the looping window was size three, 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.LoopWindowByElements(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 method `get_current_window()` will output the current window without moving the head forwards.

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

The instances of `LoopWindowByElements` 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.LoopWindowByElements(input_music,
...                                      window_size=2,
...                                      )
>>> for window in looper:
...     staff = abjad.Staff(window)
...     abjad.f(staff)
\new Staff
{
    \time 3/4
    c'4
    d'2
}
\new Staff
{
    \time 3/4
    d'2
    e'4
}
\new Staff
{
    \time 1/4
    e'4
}
```

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). 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.LoopWindowByElements(input_music,
...                                      window_size=3,
...                                      step_size=1,
...                                      max_steps=2,
...                                      repetition_chance=0.25,
...                                      head_position=0,
...                                      )
>>> looper.window_size
3
>>> looper.step_size
1
>>> looper.repetition_chance
0.25
>>> looper.max_steps
2
>>> looper.head_position
0
```

Use the `set_` methods below to change these values after initialisation.

```
>>> looper.set_window_size(2)
>>> looper.set_step_size(2)
>>> looper.set_max_steps(3)
```

(continues on next page)

(continued from previous page)

```
>>> looper.set_repetition_chance(0.1)
>>> looper.set_head_position(2)
>>> looper.window_size
2
>>> looper.step_size
2
>>> looper.max_steps
3
>>> looper.repetition_chance
1
>>> looper.head_position
2
```

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

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

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

(continues on next page)

(continued from previous page)

```
    \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.LoopWindowByElements(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
```

(continues on next page)

(continued from previous page)

```
    ~
    \time 1/16
    <e' g'>16
}
```

To change the size of the window after instantiation, use the method `set_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.LoopWindowByElements(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.set_window_size(4)
>>> notes = looper()
>>> staff = abjad.Staff(notes)
>>> abjad.f(staff)
\new Staff
{
    \time 19/8
    d'2
    e'4
    f'2
    ~
    f'8
    g'1
}
```

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.LoopWindowByElements(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")
```

(continues on next page)

(continued from previous page)

```
    \time 7/24
    d'8
    \tweak edge-height #'(0.7 . 0)
    \times 2/3 {
        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
    }
}
```

1.11 LoopWindowByList

class auxjad.**LoopWindowByList**(*container: list*, **, *window_size: int*, *step_size: int = 1*, *max_steps: int = 1*, *repetition_chance: float = 0.0*, *head_position: int = 0*)

Similar to `LoopWindowByList`, but instead of taking an `abjad.Container` input, it takes a list of arbitrary size. It then outputs the list elements, whatever they may be. The list elements can be `abjad.Container`'s, but they can also be anything else, thus being more general. Takes a list as input as well as an integer representing the number of elements per looping window, then outputs individual elements with according to the looping process. For instance, if the initial list had the elements `[A, B, C, D, E, F]` and the looping window was size three, 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.LoopWindowByList(input_list, window_size=3)
>>> looper()
['A', 'B', 'C']
>>> looper()
['B', 'C', 'D']
```

The method get_current_window() will output the current window without moving the head forwards.

```
>>> looper.get_current_window()
['B', 'C', 'D']
```

The instances of `LoopWindowByList` 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.LoopWindowByList(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). 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.LoopWindowByList(input_list,
...                                  window_size=3,
...                                  step_size=1,
...                                  max_steps=2,
...                                  repetition_chance=0.25,
...                                  head_position=0,
...                                  )
>>> looper.window_size
3
>>> looper.step_size
1
>>> looper.repetition_chance
0.25
>>> looper.max_steps
2
>>> looper.head_position
0
```

Use the `set_` methods below to change these values after initialisation.

```
>>> looper.set_window_size(2)
>>> looper.set_step_size(2)
>>> looper.set_max_steps(3)
>>> looper.set_repetition_chance(0.1)
>>> looper.set_head_position(2)
>>> looper.window_size
2
>>> looper.step_size
```

(continues on next page)

(continued from previous page)

```
2
>>> looper.max_steps
3
>>> looper.repetition_chance ==
1
>>> 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.LoopWindowByList(input_list, window_size=3)
>>> len(looper)
6
```

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

To change the size of the window after instantiation, use the method `set_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.LoopWindowByList(input_list, window_size=3)
>>> looper()
['A', 'B', 'C']
>>> looper.set_window_size(4)
>>> looper()
['B', 'C', 'D', 'E']
```

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

```
>>> input_list = [123, 'foo', (3, 4), 3.14]
>>> looper = auxjad.LoopWindowByList(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 bf2"),
...     abjad.Container(r"c''2. r4"),
... ]
>>> looper = auxjad.LoopWindowByList(input_list, window_size=3)
>>> staff = abjad.Staff()
```

(continues on next page)

(continued from previous page)

```
>>> 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
        bf2
    }
    {
        fs'1
    }
    {
        r2
        bf2
    }
    {
        c''2.
        r4
    }
    {
        r2
        bf2
    }
    {
        c''2.
        r4
    }
    {
        c''2.
        r4
    }
}
```

1.12 remove_repeated_dynamics

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

A function which removes all consecutive repeated dynamics. It removes consecutive effective dynamics, even if separated by any number of notes without one. It resets its memory of what was the previous dynamic every time it finds a hairpin, since notation such as `"c'4\f\> c'4\f\>"` is quite common; this behaviour can be toggled off using the `ignore_hairpins` keyword argument. By default, it remembers the previous dynamic even with notes separated by rests; this can be toggled off using `reset_after_rests=True`. To set a maximum length of silence after which dynamics are restated, set `reset_after_rests` to a duration using `abjad.Duration()` or any other duration format accepted by Abjad.

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

```
>>> staff = abjad.Staff(r"c'4\pp d'8\pp | c'4\f d'8\f")
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    d'8
    \pp
    c'4
    \f
    d'8
    \f
}
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    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"c'4\p d'8 | e'4.\p | c'4\p d'8\f")
>>> abjad.f(staff)
\new Staff
{
    c'4
    \p
    d'8
    e'4.
    \p
    c'4
    \p
    d'8
    \f
}
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    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
{
    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'4\pp\< d'8\f\> | c'4\f d'8\f")
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    \<
    d'8
    \f
    \>
    c'4
    \f
    d'8
    \f
}
>>> auxjad.remove_repeated_dynamics(staff)
>>> abjad.f(staff)
\new Staff
{
    c'4
```

(continues on next page)

(continued from previous page)

```
    \pp
    \<
    d'8
    \f
    \>
    c'4
    \f
    d'8
}
```

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

```
>>> staff = abjad.Staff(r"c'4\pp\< d'8\f\> | c'4\f d'8\f")
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    \<
    d'8
    \f
    \>
    c'4
    \f
    d'8
    \f
}
>>> auxjad.remove_repeated_dynamics(staff, ignore_hairpins=True)
>>> abjad.f(staff)
\new Staff
{
    c'4
    \pp
    \<
    d'8
    \f
    \>
    c'4
    d'8
}
```

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
{
    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. | 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.
    R1
```

(continues on next page)

(continued from previous page)

```
    c'4
    \pp
}
```

1.13 remove_repeated_time_signatures

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

A function which removes all unecessary time signatures. It removes consecutive effective time signatures, even if separated by any number of bars with no time signature.

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

(continues on next page)

(continued from previous page)

```
>>> 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
    }
}
>>> staff = 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
    }
}
```

1.14 simplified_time_signature_ratio

auxjad.**simplified_time_signature_ratio**(*ratio: (<class 'tuple'>, <class 'abjad.utilities.Duration.Duration'>, <class 'abjad.indicators.TimeSignature.TimeSignature'>), *, min_denominator: int = 4*) → tuple

A function simplifies the ratio of a given time signature respecting a minimum denominator value. Input is a tuple of two integers.

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.

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

If a ratio cannot be simplified at all, the function returns the original values.

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

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

1.15 sync_containers

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

Takes an arbitrary number of `abjad.Container`'s (or child classes), finds the longest one 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)
{
    %%% \time 4/4 %%%
    g'2.
```

(continues on next page)

(continued from previous page)

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

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 %%%
```

(continues on next page)

(continued from previous page)

```
    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,
...                        )
>>> 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
```

(continues on next page)

(continued from previous page)

```
    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(
```

(continues on next page)

(continued from previous page)

```
...     r'''
...     \context {
...         \Score
...         \remove "Timing_translator"
...         \remove "Default_bar_line_engraver"
...     }
...     \context {
...         \Staff
...         \consists "Timing_translator"
...         \consists "Default_bar_line_engraver"
...     }
...     ''')
>>> abjad.show(lilypond_file)
```

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
```

1.16 TenneysContainer

class auxjad.**TenneysContainer**(*container: list*, *, *weights: list = None*, *curvature: float = 1.0*)

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

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

Applying the `len()` function to the container will give the length of the container.

```
>>> len(container)
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).

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

Calling the container will output one of its elements, selected according to the current probability values. 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 += container()
>>> 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.

```
>>> container.probabilities
[6.0, 5.0, 0.0, 3.0, 1.0, 2.0]
>>> container.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):

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

Using a convex curvature:

```
>>> container = auxjad.TenneysContainer(['A', 'B', 'C', 'D', 'E', 'F'],
...                                     curvature=0.2,
...                                     )
>>> container.curvature
```

(continues on next page)

(continued from previous page)

```
0.2
>>> container.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container()
'C'
>>> container.curvature
0.2
>>> container.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container.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 += container()
>>> result
'DACBEDFACABDACECBEFAEDBAFBABFD'
```

Checking the probability values at this point outputs:

```
>>> container.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:

```
>>> container = auxjad.TenneysContainer(['A', 'B', 'C', 'D', 'E', 'F'],
...                                     curvature=15.2,
...                                     )
>>> container.curvature
0.2
>>> container.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container.probabilities
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container()
'C'
>>> container.curvature
0.2
>>> container.weights
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
>>> container.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 += container()
>>> result
'DFAECBDFAECBDFAECBDFAECBDFAECB'
```

Checking the probability values at this point outputs:

```
>>> container.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.

```
>>> container = auxjad.TenneysContainer(
...     ['A', 'B', 'C', 'D', 'E', 'F'],
...     weights=[1.0, 1.0, 5.0, 5.0, 10.0, 20.0],
>>> )
>>> container.weights
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> container.probabilities
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> result = ''
>>> for _ in range(30):
...     result += container()
>>> result
'FBEFECFDEADFEDFEDBFECDAFCEDCFE'
>>> container.weights
[1.0, 1.0, 5.0, 5.0, 10.0, 20.0]
>>> container.probabilities
[7.0, 12.0, 10.0, 15.0, 0.0, 20.0]
```

To reset the probability to its initial value, use the method `reset_probabilities()`.

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

To replace an element in the container, use the method `replace_element()`. This will not affect the current probability vector, and the new element will have the same probability as the one it replaced.

```
>>> container = auxjad.TenneysContainer(['A', 'B', 'C', 'D', 'E', 'F'])
>>> for _ in range(30):
```

(continues on next page)

(continued from previous page)

```
...     container()
>>> container.replace_element('foo', 2)
>>> container.contents
['A', 'B', 'foo', 'D', 'E', 'F']
>>> container.probabilities
[3.0, 2.0, 1.0, 7.0, 5.0, 0.0]
```

A new container of an arbitrary length can be set at any point using the method `set_container()`. 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.

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

To change the curvature value at any point, use the `set_curvature()` method.

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

1.17 underfull_duration

`auxjad.`**`underfull_duration`**(*container: abjad.core.Container.Container*) → abjad.utilities.Duration.Duration

Checks if an `abjad.Container` is underfull and returns the missing `abjad.Duration`.

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

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
```

PYTHON MODULE INDEX

a

INDEX