mutwo API documentation

mutwo.abjad_converters

Table of content

Build Lilypond scores via Abjad from Mutwo data.

The following converter classes help to quantize and translate Mutwo data to Western notation. Due to the complex nature of this task, Mutwo tries to offer as many optional arguments as possible through which the user can affect the conversion routines. The most important class and best starting point for organising a conversion setting is `SequentialEventToAbjadVoiceConverter`. If one wants to build complete scores from within mutwo, the module offers the `NestedComplexEventToAbjadContainerConverter`.

Known bugs and limitations:

1. Indicators attached to rests which follow another rest won't be translated to *abjad*. This behaviour happens because the `SequentialEventToAbjadVoiceConverter` ties rests before converting the data to *abjad* objects.
2. Quantization can be slow and not precise. Try both quantization classes. Change the parameters. Use different settings and classes for different parts of your music.

Object	Documentation
mutwo.abjad_converters.SequentialEventToQuantizedAbjadContainer	Quantize *SequentialEvent* objects.
mutwo.abjad_converters.NauertSequentialEventToQuantizedAbjadContainer	Quantize *SequentialEvent* objects via `abjadext.nauert`.
mutwo.abjad_converters.NauertSequentialEventToDurationLineBasedQuantizedAbjadContainer	Quantize *SequentialEvent* objects via `abjadext.nauert`.
mutwo.abjad_converters.LeafMakerSequentialEventToQuantizedAbjadContainer	Quantize *SequentialEvent* object via `abjad.LeafMaker`.
mutwo.abjad_converters.LeafMakerSequentialEventToDurationLineBasedQuantizedAbjadContainer	Quantize *SequentialEvent* object via `abjad.LeafMaker`.
mutwo.abjad_converters.ComplexEventToAbjadContainer	
mutwo.abjad_converters.SequentialEventToAbjadVoice	Convert *SequentialEvent* to `abjad.Voice`.
mutwo.abjad_converters.NestedComplexEventToAbjadContainer	
mutwo.abjad_converters.NestedComplexEventToComplexEventToAbjadContainers	
mutwo.abjad_converters.CycleBasedNestedComplexEventToComplexEventToAbjadContainers	
mutwo.abjad_converters.TagBasedNestedComplexEventToComplexEventToAbjadContainers	
mutwo.abjad_converters.MutwoLyricToAbjadString	
mutwo.abjad_converters.MutwoPitchToAbjadPitch	Convert Mutwo Pitch objects to Abjad Pitch objects.
mutwo.abjad_converters.TempoEnvelopeToAbjadAttachmentTempo	Convert tempo envelope to `Tempo`.
mutwo.abjad_converters.ComplexTempoEnvelopeToAbjadAttachmentTempo	Convert tempo envelope to `Tempo`.
mutwo.abjad_converters.MutwoVolumeToAbjadAttachmentDynamic	Convert Mutwo Volume objects to `Dynamic`.
mutwo.abjad_converters.MutwoPitchToHEJIAbjadPitch	Convert Mutwo `JustIntonationPitch` objects to Abjad Pitch objects.
mutwo.abjad_converters.ProcessAbjadContainerRoutine	
mutwo.abjad_converters.AddDurationLineEngraver	
mutwo.abjad_converters.PrepareForDurationLineBasedNotation	
mutwo.abjad_converters.AddInstrumentName	
mutwo.abjad_converters.AddAccidentalStyle	
mutwo.abjad_converters.SetStaffSize	

class SequentialEventToQuantizedAbjadContainer(*time_signature_sequence=(TimeSignature((4, 4)),), tempo_envelope=None*)

Bases: *Converter*

Quantize *SequentialEvent* objects.

Parameters

- `time_signature_sequence` (*`Sequence[TimeSignature]`*) – Set time signatures to divide the quantized abjad data in desired bar sizes. If the converted *`SequentialEvent`* is longer than the sum of all passed time signatures, the last time signature will be repeated for the remaining bars.
- `tempo_envelope` (`TempoEnvelope`) – Defines the tempo of the converted music. This is an `core_events.TempoEnvelope` object which durations are beats and which levels are either numbers (that will be interpreted as beats per minute ('BPM')) or *`TempoPoint`* objects. If no tempo envelope has been defined, Mutwo will assume a constant tempo of 1/4 = 120 BPM.

`abstract convert`(*sequential_event_to_convert*)

Parameters
`sequential_event_to_convert` (`SequentialEvent`) –

Return type
tuple[abjad.score.Container, tuple[tuple[tuple[int, ...], ...], ...]]

`property tempo_envelope:` *`TempoEnvelope`*

`class NauertSequentialEventToQuantizedAbjadContainer`(*time_signature_sequence=(TimeSignature((4, 4)),), duration_unit='beats', tempo_envelope=None, attack_point_optimizer=<abjadext.nauert.attackpointoptimizers.MeasurewiseAttackPointOptimizer object>, search_tree=None*)

Bases: *`SequentialEventToQuantizedAbjadContainer`*

Quantize *`SequentialEvent`* objects via `abjadext.nauert`.

Parameters

- `time_signature_sequence` (*`Sequence[TimeSignature]`*) – Set time signatures to divide the quantized abjad data in desired bar sizes. If the converted *`SequentialEvent`* is longer than the sum of all passed time signatures, the last time signature will be repeated for the remaining bars.
- `duration_unit` (*`str`*) – This defines the *duration_unit* of the passed *`SequentialEvent`* (how the `duration` attribute will be interpreted). Can either be 'beats' (default) or 'miliseconds'. WARNING: 'miliseconds' isn't working properly yet!
- `tempo_envelope` (`TempoEnvelope`) – Defines the tempo of the converted music. This is an `core_events.TempoEnvelope` object which durations are beats and which levels are either numbers (that will be interpreted as beats per minute ('BPM')) or *`TempoPoint`* objects. If no tempo envelope has been defined, Mutwo will assume a constant tempo of 1/4 = 120 BPM.
- `attack_point_optimizer` (*`Optional[AttackPointOptimizer]`*) – Optionally the user can pass a `nauert.AttackPointOptimizer` object. Attack point optimizer help to split events and tie them for better looking notation. The default attack point optimizer is `nauert.MeasurewiseAttackPointOptimizer` which splits events to better represent metrical structures within bars. If no optimizer is desired this argument can be set to `None`.
- `search_tree` (*`Optional[SearchTree]`*) –

Unlike *`LeafMakerSequentialEventToQuantizedAbjadContainer`* this converter supports nested tuplets and ties across tuplets. But this converter is much slower than the *`LeafMakerSequentialEventToQuantizedAbjadContainer`*. Because the converter depends on the abjad extension *nauert* its quality is dependent on the inner mechanism of the used package. Because the quantization made by the *nauert* package can be somewhat indeterministic a lot of tweaking may be necessary for complex musical structures.

`convert`(*sequential_event_to_convert*)

Parameters
`sequential_event_to_convert` (`SequentialEvent`) –

Return type
tuple[abjad.score.Container, tuple[tuple[tuple[int, ...], ...], ...]]

`class NauertSequentialEventToDurationLineBasedQuantizedAbjadContainer`(**args, duration_line_minimum_length=6, duration_line_thickness=3, **kwargs*)

Bases: *`NauertSequentialEventToQuantizedAbjadContainer`*, `_DurationLineBasedQuantizedAbjadContainerMixin`

Quantize *`SequentialEvent`* objects via `abjadext.nauert`.

Parameters

- `time_signature_sequence` – Set time signatures to divide the quantized abjad data in desired bar sizes. If the converted *`SequentialEvent`* is longer than the sum of all passed time signatures, the last time signature will be repeated for the remaining bars.
- `duration_unit` – This defines the *duration_unit* of the passed *`SequentialEvent`* (how the `duration` attribute will be interpreted). Can either be 'beats' (default) or 'miliseconds'. WARNING: 'miliseconds' isn't working properly yet!

- **tempo_envelope** – Defines the tempo of the converted music. This is an `core_events.TempoEnvelope` object which durations are beats and which levels are either numbers (that will be interpreted as beats per minute ('BPM')) or *TempoPoint* objects. If no tempo envelope has been defined, Mutwo will assume a constant tempo of 1/4 = 120 BPM.
- **attack_point_optimizer** – Optionally the user can pass a `nauert.AttackPointOptimizer` object. Attack point optimizer help to split events and tie them for better looking notation. The default attack point optimizer is `nauert.MeasurewiseAttackPointOptimizer` which splits events to better represent metrical structures within bars. If no optimizer is desired this argument can be set to `None`.
- **duration_line_minimum_length** (*int*) – The minimum length of a duration line.
- **duration_line_thickness** (*int*) – The thickness of a duration line.

This converter differs from its parent class through the usage of duration lines for indicating rhythm instead of using flags, beams, dots and note head colors.

Note:

Don't forget to add the 'Duration_line_engraver' to the resulting abjad Voice, otherwise Lilypond won't be able to render the desired output.

Example:

```
>>> import abjad
>>> from mutwo import abjad_converters
>>> from mutwo import core_events
>>> converter = abjad_converters.SequentialEventToAbjadVoiceConverter(
>>>     abjad_converters.LeafMakerSequentialEventToDurationLineBasedQuantizedAbjadContainer(
>>>         )
>>>     )
>>> sequential_event_to_convert = core_events.SequentialEvent(
>>>     [
>>>         music_events.NoteLike("c", 0.125),
>>>         music_events.NoteLike("d", 1),
>>>         music_events.NoteLike([], 0.125),
>>>         music_events.NoteLike("e", 0.16666),
>>>         music_events.NoteLike("e", 0.08333333333333333)
>>>     ]
>>> )
>>> converted_sequential_event = converter.convert(sequential_event_to_convert)
>>> converted_sequential_event.consists_commands.append("Duration_line_engraver")
```

`convert`(*sequential_event_to_convert*)

Parameters
`sequential_event_to_convert` (`SequentialEvent`) –

Return type
tuple[abjad.score.Container, tuple[tuple[tuple[int, ...], ...], ...]]

`class LeafMakerSequentialEventToQuantizedAbjadContainer`(**args*, *do_rewrite_meter=True*, *add_beams=True*, ***kwargs*)

Bases: *SequentialEventToQuantizedAbjadContainer*

Quantize *SequentialEvent* object via `abjad.LeafMaker`.

Parameters

- **time_signature_sequence** – Set time signatures to divide the quantized abjad data in desired bar sizes. If the converted *SequentialEvent* is longer than the sum of all passed time signatures, the last time signature will be repeated for the remaining bars.
- **tempo_envelope** – Defines the tempo of the converted music. This is an `core_events.TempoEnvelope` object which durations are beats and which levels are either numbers (that will be interpreted as beats per minute ('BPM')) or *TempoPoint* objects. If no tempo envelope has been defined, Mutwo will assume a constant tempo of 1/4 = 120 BPM.
- **do_rewrite_meter** (*bool*) –
- **add_beams** (*bool*) –

This method is significantly faster than the *NauertSequentialEventToQuantizedAbjadContainer*. But it also has several known limitations:

1. *LeafMakerSequentialEventToQuantizedAbjadContainer* doesn't support nested tuplets.

2. *LeafMakerSequentialEventToQuantizedAbjadContainer* doesn't support ties across tuplets with different prolation (or across tuplets and not-tuplet notation). If ties are desired the user has to build them manually before passing the *SequentialEvent* to the converter.

`convert`(*sequential_event_to_convert*)

Parameters
sequential_event_to_convert (SequentialEvent) –

Return type
tuple[abjad.score.Container, tuple[tuple[tuple[int, ...], ...], ...]]

class LeafMakerSequentialEventToDurationLineBasedQuantizedAbjadContainer(**args*, *duration_line_minimum_length=6*, *duration_line_thickness=3*, ***kwargs*)

Bases: *LeafMakerSequentialEventToQuantizedAbjadContainer*, `_DurationLineBasedQuantizedAbjadContainerMixin`

Quantize *SequentialEvent* object via `abjad.LeafMaker`.

Parameters

- **time_signature_sequence** – Set time signatures to divide the quantized abjad data in desired bar sizes. If the converted *SequentialEvent* is longer than the sum of all passed time signatures, the last time signature will be repeated for the remaining bars.
- **tempo_envelope** – Defines the tempo of the converted music. This is an `core_events.TempoEnvelope` object which durations are beats and which levels are either numbers (that will be interpreted as beats per minute ('BPM')) or *TempoPoint* objects. If no tempo envelope has been defined, Mutwo will assume a constant tempo of 1/4 = 120 BPM.
- **duration_line_minimum_length** (*int*) – The minimum length of a duration line.
- **duration_line_thickness** (*int*) – The thickness of a duration line.

This converter differs from its parent class through the usage of duration lines for indicating rhythm instead of using flags, beams, dots and note head colors.

Note:

Don't forget to add the 'Duration_line_engraver' to the resulting abjad Voice, otherwise Lilypond won't be able to render the desired output.

Example:

```
>>> import abjad
>>> from mutwo import abjad_converters
>>> from mutwo import core_events
>>> converter = abjad_converters.SequentialEventToAbjadVoiceConverter(
>>>     abjad_converters.LeafMakerSequentialEventToDurationLineBasedQuantizedAbjadContainer(
>>>        )
>>>    )
>>> sequential_event_to_convert = core_events.SequentialEvent(
>>>     [
>>>         music_events.NoteLike("c", 0.125),
>>>         music_events.NoteLike("d", 1),
>>>         music_events.NoteLike([], 0.125),
>>>         music_events.NoteLike("e", 0.16666),
>>>         music_events.NoteLike("e", 0.08333333333333333)
>>>     ]
>>> )
>>> converted_sequential_event = converter.convert(sequential_event_to_convert)
>>> converted_sequential_event.consists_commands.append("Duration_line_engraver")
```

`convert`(*sequential_event_to_convert*)

Parameters
sequential_event_to_convert (SequentialEvent) –

Return type
tuple[abjad.score.Container, tuple[tuple[tuple[int, ...], ...], ...]]

class ComplexEventToAbjadContainer(*abjad_container_class*, *lilypond_type_of_abjad_container*, *complex_event_to_abjad_container_name*, *pre_process_abjad_container_routine_sequence*, *post_process_abjad_container_routine_sequence*)

Bases: *Converter*

Parameters

- **`abjad_container_class`** (*`Type[Container]`*) –
- **`lilypond_type_of_abjad_container`** (*`str`*) –
- **`complex_event_to_abjad_container_name`** (*`Callable[[ComplexEvent], str]`*) –
- **`pre_process_abjad_container_routine_sequence`** (*`Sequence[ProcessAbjadContainerRoutine]`*) –
- **`post_process_abjad_container_routine_sequence`** (*`Sequence[ProcessAbjadContainerRoutine]`*) –

`convert`(*complex_event_to_convert*)

Parameters
`complex_event_to_convert` (`ComplexEvent`) –

Return type
Container

class `SequentialEventToAbjadVoice`(*sequential_event_to_quantized_abjad_container=<mutwo.abjad_converters.events.quantization.NauertSequentialEventToQuantizedAbjadContainer object>, simple_event_to_pitch_list=<mutwo.music_converters.parsers.SimpleEventToPitchList object>, simple_event_to_volume=<mutwo.music_converters.parsers.SimpleEventToVolume object>, simple_event_to_grace_note_sequential_event=<mutwo.music_converters.parsers.SimpleEventToGraceNoteSequentialEvent object>, simple_event_to_after_grace_note_sequential_event=<mutwo.music_converters.parsers.SimpleEventToAfterGraceNoteSequentialEvent object>, simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPlayingIndicatorCollection object>, simple_event_to_notation_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToNotationIndicatorCollection object>, simple_event_to_lyric=<mutwo.music_converters.parsers.SimpleEventToLyric object>, is_simple_event_rest=None, mutwo_pitch_to_abjad_pitch=<mutwo.abjad_converters.parameters.pitches.MutwoPitchToAbjadPitch object>, mutwo_volume_to_abjad_attachment_dynamic=<mutwo.abjad_converters.parameters.volumes.MutwoVolumeToAbjadAttachmentDynamic object>, tempo_envelope_to_abjad_attachment_tempo=<mutwo.abjad_converters.parameters.tempos.ComplexTempoEnvelopeToAbjadAttachmentTempo object>, mutwo_lyric_to_abjad_string=<mutwo.abjad_converters.parameters.lyrics.MutwoLyricToAbjadString object>, abjad_attachment_class_sequence=None, write_multimeasure_rests=True, abjad_container_class=<class 'abjad.score.Voice'>, lilypond_type_of_abjad_container='Voice', complex_event_to_abjad_container_name=<function SequentialEventToAbjadVoice.<lambda>>, pre_process_abjad_container_routine_sequence=(), post_process_abjad_container_routine_sequence=()*)

Bases: *`ComplexEventToAbjadContainer`*

Convert *`SequentialEvent`* to `abjad.Voice`.

Parameters

- **`sequential_event_to_quantized_abjad_container`** (`SequentialEventToQuantizedAbjadContainer`, *`optional`*) – Class which defines how the Mutwo data will be quantized. See *`SequentialEventToQuantizedAbjadContainer`* for more information.
- **`simple_event_to_pitch_list`** (*`Callable[[`*`core_events.SimpleEvent`*`]`*, `music_parameters.abc.Pitch`*`]`*, *`optional`*) – Function to extract from a *`mutwo.core_events.SimpleEvent`* a tuple that contains pitch objects (objects that inherit from *`mutwo.music_parameters.abc.Pitch`*). By default it asks the Event for its `pitch_list` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their pitch property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no pitch can be extracted), mutwo will assume an event without any pitches.
- **`simple_event_to_volume`** (*`Callable[[`*`core_events.SimpleEvent`*`]`*, `music_parameters.abc.Volume`*`]`*, *`optional`*) – Function to extract the volume from a *`mutwo.core_events.SimpleEvent`* in the purpose of generating dynamic indicators. The function should return an object that inherits from *`mutwo.music_parameters.abc.Volume`*. By default it asks the Event for its `volume` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their volume property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no volume can be extracted), mutwo will set `pitch_list` to an empty list and set volume to 0.
- **`simple_event_to_grace_note_sequential_event`** (*`Callable[[`*`core_events.SimpleEvent`*`]`*, `core_events.SequentialEvent`*`[`*`core_events.SimpleEvent`*`]]`*, *`optional`*) – Function to extract from a *`mutwo.core_events.SimpleEvent`* a *`SequentialEvent`* object filled with *`SimpleEvent`*. By default it asks the Event for its `grace_note_sequential_event` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their *grace_note_sequential_event* property, this

argument should be overridden. If the function call raises an `AttributeError` (e.g. if no grace_note_sequential_event can be extracted), mutwo will use an empty *SequentialEvent*.

- **simple_event_to_after_grace_note_sequential_event** (*Callable[[core_events.SimpleEvent], core_events.SequentialEvent[core_events.SimpleEvent]], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *SequentialEvent* object filled with *SimpleEvent*. By default it asks the Event for its `after_grace_note_sequential_event` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their *after_grace_note_sequential_event* property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no after_grace_note_sequential_event can be extracted), mutwo will use an empty *SequentialEvent*.
- **simple_event_to_playing_indicator_collection** (*Callable[[core_events.SimpleEvent], music_parameters.PlayingIndicatorCollection,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.playing_indicators.PlayingIndicatorCollection* object. By default it asks the Event for its `playing_indicator_collection` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their playing_indicators property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.
- **simple_event_to_notation_indicator_collection** (*Callable[[core_events.SimpleEvent], music_parameters.NotationIndicatorCollection,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.notation_indicators.NotationIndicatorCollection* object. By default it asks the Event for its `notation_indicators` (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their playing_indicators property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no notation indicator collection can be extracted), mutwo will build a notation indicator collection from *DEFAULT_NOTATION_INDICATORS_COLLECTION_CLASS*
- **simple_event_to_lyric** (*Callable[[core_events.SimpleEvent], music_parameters.abc.Lyric], optional*) – Function to extract the lyric from a *mutwo.core_events.SimpleEvent* in the purpose of generating lyrics. The function should return an object that inherits from *mutwo.music_parameters.abc.Lyric*. By default it asks the Event for its `lyric` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their lyric property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no lyric can be extracted), mutwo will set `lyric` to an empty text.
- **is_simple_event_rest** (*Callable[[core_events.SimpleEvent], bool], optional*) – Function to detect if the the inspected *mutwo.core_events.SimpleEvent* is a Rest. By default Mutwo simply checks if 'pitch_list' contain any objects. If not, the Event will be interpreted as a rest.
- **mutwo_pitch_to_abjad_pitch** (*MutwoPitchToAbjadPitch, optional*) – Class which defines how to convert *mutwo.music_parameters.abc.Pitch* objects to `abjad.Pitch` objects. See *MutwoPitchToAbjadPitch* for more information.
- **mutwo_volume_to_abjad_attachment_dynamic** (*MutwoVolumeToAbjadAttachmentDynamic, optional*) – Class which defines how to convert *mutwo.music_parameters.abc.Volume* objects to `mutwo.converters.frontends.abjad_parameters.Dynamic` objects. See *MutwoVolumeToAbjadAttachmentDynamic* for more information.
- **tempo_envelope_to_abjad_attachment_tempo** (*TempoEnvelopeToAbjadAttachmentTempo, optional*) – Class which defines how to convert tempo envelopes to `mutwo.converters.frontends.abjad_parameters.Tempo` objects. See *TempoEnvelopeToAbjadAttachmentTempo* for more information.
- **mutwo_lyric_to_abjad_string** (*MutwoLyricToAbjadString*) – Callable which defines how to convert *mutwo.music_parameters.abc.Lyric* to a string. Consult *mutwo.abjad_converters.MutwoLyricToAbjadString* for more information.
- **abjad_attachment_class_sequence** (*Sequence[abjad_parameters.abc.AbjadAttachment], optional*) – A tuple which contains all available abjad attachment classes which shall be used by the converter.
- **write_multimeasure_rests** (*bool*) – Set to `True` if the converter should replace rests that last a complete bar with multimeasure rests (rests with uppercase "R" in Lilypond). Default to `True`.
- **abjad_container_class** (*Type[Container]*) –
- **lilypond_type_of_abjad_container** (*str*) –
- **complex_event_to_abjad_container_name** (*Callable[[ComplexEvent], Optional[str]]*) –
- **pre_process_abjad_container_routine_sequence** (*Sequence[ProcessAbjadContainerRoutine]*) –
- **post_process_abjad_container_routine_sequence** (*Sequence[ProcessAbjadContainerRoutine]*) –

ExtractedData

alias of tuple[list[*Pitch*], *Volume*, *SequentialEvent*[*SimpleEvent*], *SequentialEvent*[*SimpleEvent*], *PlayingIndicatorCollection*, *NotationIndicatorCollection*, *Lyric*]

ExtractedDataPerSimpleEvent

alias of tuple[tuple[list[*Pitch*], *Volume*, *SequentialEvent*[*SimpleEvent*], *SequentialEvent*[*SimpleEvent*], *PlayingIndicatorCollection*, *NotationIndicatorCollection*, *Lyric*], ...]

convert(*sequential_event_to_convert*)

Convert passed *SequentialEvent*.

Parameters

sequential_event_to_convert (mutwo.core_events.SequentialEvent) – The *SequentialEvent* which shall be converted to the abjad.Voice object.

Return type

Voice

Example:

```
>>> import abjad
>>> from mutwo.events import basic, music
>>> from mutwo.converters.frontends import abjad as mutwo_abjad
>>> mutwo_melody = basic.SequentialEvent(
>>>     [
>>>         music.NoteLike(pitch, duration)
>>>         for pitch, duration in zip("c a g e".split(" "), (1, 1 / 6, 1 / 6, 1 / 6))
>>>     ]
>>> )
>>> converter = mutwo_abjad.SequentialEventToAbjadVoice()
>>> abjad_melody = converter.convert(mutwo_melody)
>>> abjad.lilypond(abjad_melody)
\new Voice
{
    {
        \tempo 4=120
        %%% \time 4/4 %%%
        c'1
        \mf
    }
    {
        \times 2/3 {
            a'4
            g'4
            e'4
        }
        r2
    }
}
```

class NestedComplexEventToAbjadContainer(*nested_complex_event_to_complex_event_to_abjad_container_converters_converter, abjad_container_class, lilypond_type_of_abjad_container, complex_event_to_abjad_container_name=<function NestedComplexEventToAbjadContainer.<lambda>>, pre_process_abjad_container_routine_sequence=(), post_process_abjad_container_routine_sequence=()*)

Bases: *ComplexEventToAbjadContainer*

Parameters

- **nested_complex_event_to_complex_event_to_abjad_container_converters_converter** (NestedComplexEventToComplexEventToAbjadContainers) –
- **abjad_container_class** (*Type[Container]*) –
- **lilypond_type_of_abjad_container** (*str*) –
- **complex_event_to_abjad_container_name** (*Callable[[*ComplexEvent*], str]*) –

- **pre_process_abjad_container_routine_sequence** (*Sequence[ProcessAbjadContainerRoutine]*) –
- **post_process_abjad_container_routine_sequence** (*Sequence[ProcessAbjadContainerRoutine]*) –

`class NestedComplexEventToComplexEventToAbjadContainers`

Bases: *Converter*

`abstract convert`(*nested_complex_event_to_convert*)

Parameters
nested_complex_event_to_convert (ComplexEvent) –

Return type
tuple[*mutwo.abjad_converters.events.building.ComplexEventToAbjadContainer*, ...]

`class CycleBasedNestedComplexEventToComplexEventToAbjadContainers`(*complex_event_to_abjad_container_converter_sequence*)

Bases: *NestedComplexEventToComplexEventToAbjadContainers*

Parameters
complex_event_to_abjad_container_converter_sequence (*Sequence[ComplexEventToAbjadContainer]*) –

`convert`(*nested_complex_event_to_convert*)

Parameters
nested_complex_event_to_convert (ComplexEvent) –

Return type
tuple[*mutwo.abjad_converters.events.building.ComplexEventToAbjadContainer*, ...]

`class TagBasedNestedComplexEventToComplexEventToAbjadContainers`(*tag_to_abjad_converter_dict*, *complex_event_to_tag=<function TagBasedNestedComplexEventToComplexEventToAbjadContainers.<lambda>>*)

Bases: *NestedComplexEventToComplexEventToAbjadContainers*

Parameters

- **tag_to_abjad_converter_dict** (*dict[str, mutwo.abjad_converters.events.building.ComplexEventToAbjadContainer]*) –
- **complex_event_to_tag** (*Callable[[ComplexEvent], str]*) –

`convert`(*nested_complex_event_to_convert*)

Parameters
nested_complex_event_to_convert (ComplexEvent) –

Return type
tuple[*mutwo.abjad_converters.events.building.ComplexEventToAbjadContainer*, ...]

`class MutwoLyricToAbjadString`

Bases: *Converter*

`convert`(*mutwo_lyric_to_convert*)

Parameters
mutwo_lyric_to_convert (Lyric) –

Return type
str

`class MutwoPitchToAbjadPitch`

Bases: *Converter*

Convert Mutwo Pitch objects to Abjad Pitch objects.

This default class simply checks if the passed Mutwo object belongs to `mutwo.ext.parameters.pitches.WesternPitch`. If it does, Mutwo will initialise the Abjad Pitch from the `name` attribute. Otherwise Mutwo will simply initialise the Abjad Pitch from the objects `frequency` attribute.

If users desire to make more complex conversions (for instance due to `scordatura` or transpositions of instruments), one can simply inherit from this class to define more complex cases.

`convert`(*pitch_to_convert*)

Parameters
pitch_to_convert (Pitch) –

Return type
Pitch

`class TempoEnvelopeToAbjadAttachmentTempo`

Bases: *Converter*

Convert tempo envelope to `Tempo`.

Abstract base class for tempo envelope conversion. See *ComplexTempoEnvelopeToAbjadAttachmentTempo* for a concrete class.

`abstract convert`(*tempo_envelope_to_convert*)

Parameters
`tempo_envelope_to_convert` (TempoEnvelope) –

Return type
tuple[tuple[*Union*[float, fractions.Fraction, int], *mutwo.abjad_parameters.attachments.Tempo*], ...]

`class ComplexTempoEnvelopeToAbjadAttachmentTempo`

Bases: *TempoEnvelopeToAbjadAttachmentTempo*

Convert tempo envelope to `Tempo`.

This object tries to intelligently set correct tempo abjad_parameters to an `abjad.Voice` object, appropriate to Western notation standards. Therefore it will not repeat tempo indications if they are merely repetitions of previous tempo indications and it will write 'a tempo' when returning to the same tempo after ritardandi or accelerandi.

`convert`(*tempo_envelope_to_convert*)

Parameters
`tempo_envelope_to_convert` (TempoEnvelope) –

Return type
tuple[tuple[*Union*[float, fractions.Fraction, int], *mutwo.abjad_parameters.attachments.Tempo*], ...]

`class MutwoVolumeToAbjadAttachmentDynamic`

Bases: *Converter*

Convert Mutwo Volume objects to `Dynamic`.

This default class simply checks if the passed Mutwo object belongs to `mutwo.ext.parameters.volumes.WesternVolume`. If it does, Mutwo will initialise the `Tempo` object from the `name` attribute. Otherwise Mutwo will first initialise a `WesternVolume` object via its py:method:*mutwo.ext.parameters.volumes.WesternVolume.from_amplitude* method.

Hairpins aren't notated with the aid of `mutwo.ext.parameters.abc.Volume` objects, but with `mutwo.ext.parameters.playing_indicators.Hairpin`.

`convert`(*volume_to_convert*)

Parameters
`volume_to_convert` (Volume) –

Return type
Optional[Dynamic]

`class MutwoPitchToHEJIAbjadPitch`(*reference_pitch='a', prime_to_heji_accidental_name=None, otonality_indicator=None, utonality_indicator=None, exponent_to_exponent_indicator=None, tempered_pitch_indicator=None*)

Bases: *MutwoPitchToAbjadPitch*

Convert Mutwo `JustIntonationPitch` objects to Abjad Pitch objects.

Parameters

- `reference_pitch` (*str, optional*) – The reference pitch (1/1). Should be a diatonic pitch name (see `DIATONIC_PITCH_CLASS_CONTAINER`) in English nomenclature. For any other reference pitch than 'c', Lilyponds midi rendering for pitches with the diatonic pitch 'c' will be slightly out of tune (because the first value of **:arg:`global_scale`** always have to be 0).
- `prime_to_heji_accidental_name` (*dict[int, str], optional*) – Mapping of a prime number to a string which indicates the respective prime number in the resulting accidental name. See *mutwo.ekmelily_converters.configurations.DEFAULT_PRIME_TO_HEJI_ACCIDENTAL_NAME_DICT* for the default mapping.
- `otonality_indicator` (*str, optional*) – String which indicates that the respective prime alteration is otonal. See *mutwo.ekmelily_converters.configurations.DEFAULT_OTONALITY_INDICATOR* for the default value.

- **utonality_indicator** (*str, optional*) – String which indicates that the respective prime alteration is utonal. See *mutwo.ekmelily_converters.configurations.DEFAULT_OTONALITY_INDICATOR* for the default value.
- **exponent_to_exponent_indicator** (*Callable[[int], str], optional*) – Function to convert the exponent of a prime number to string which indicates the respective exponent. See *mutwo.ekmelily_converters.configurations.DEFAULT_EXPONENT_TO_EXPONENT_INDICATOR()* for the default function.
- **tempered_pitch_indicator** (*str, optional*) – String which indicates that the respective accidental is tempered (12 EDO). See *mutwo.ekmelily_converters.configurations.DEFAULT_TEMPERED_PITCH_INDICATOR* for the default value.

The resulting Abjad pitches are expected to be used in combination with tuning files that are generated by `HEJIEkmelilyTuningFileConverter` and with the Lilypond extension Ekmelily. You can find pre-generated tuning files here.

Example:

```
>>> from mutwo.ext.parameters import pitches
>>> from mutwo.converters.frontends import abjad
>>> my_ji_pitch = pitches.JustIntonationPitch('5/4')
>>> converter_on_a = abjad.MutwoPitchToHEJIAbjadPitch(reference_pitch='a')
>>> converter_on_c = abjad.MutwoPitchToHEJIAbjadPitch(reference_pitch='c')
>>> converter_on_a.convert(my_ji_pitch)
NamedPitch("csoaa''")
>>> converter_on_c.convert(my_ji_pitch)
NamedPitch("eoaa'")
```

convert(*pitch_to_convert*)

Parameters
pitch_to_convert (Pitch) –

Return type
Pitch

class ProcessAbjadContainerRoutine

Bases: ABC

class AddDurationLineEngraver

Bases: *ProcessAbjadContainerRoutine*

class PrepareForDurationLineBasedNotation

Bases: *ProcessAbjadContainerRoutine*

class AddInstrumentName(*complex_event_to_instrument_name=<function AddInstrumentName.<lambda>>, complex_event_to_short_instrument_name=<function AddInstrumentName.<lambda>>, instrument_name_font_size='teeny', short_instrument_name_font_size='teeny'*)

Bases: *ProcessAbjadContainerRoutine*

Parameters

- **complex_event_to_instrument_name** (*Callable[[*ComplexEvent*], str]*) –
- **complex_event_to_short_instrument_name** (*Callable[[*ComplexEvent*], str]*) –
- **instrument_name_font_size** (*str*) –
- **short_instrument_name_font_size** (*str*) –

class AddAccidentalStyle(*accidental_style*)

Bases: *ProcessAbjadContainerRoutine*

Parameters
accidental_style (*str*) –

class SetStaffSize(*difference_of_size*)

Bases: *ProcessAbjadContainerRoutine*

Parameters
difference_of_size (*int*) –

mutwo.abjad_converters.configurations

Configure *mutwo.abjad_converters*.

DEFAULT_ABJAD_ATTACHMENT_CLASS_TUPLE = (<class 'mutwo.abjad_parameters.attachments.AfterGraceNoteSequentialEvent'>, <class 'mutwo.abjad_parameters.attachments.Arpeggio'>, <class 'mutwo.abjad_parameters.attachments.Articulation'>, <class 'mutwo.abjad_parameters.attachments.ArtificalHarmonic'>, <class 'mutwo.abjad_parameters.attachments.BarLine'>, <class 'mutwo.abjad_parameters.attachments.BartokPizzicato'>, <class 'mutwo.abjad_parameters.attachments.BendAfter'>, <class 'mutwo.abjad_parameters.attachments.BreathMark'>, <class 'mutwo.abjad_parameters.attachments.Clef'>, <class 'mutwo.abjad_parameters.attachments.Cue'>, <class 'mutwo.abjad_parameters.attachments.DurationLineDashed'>, <class 'mutwo.abjad_parameters.attachments.DurationLineTriller'>, <class 'mutwo.abjad_parameters.attachments.Dynamic'>, <class 'mutwo.abjad_parameters.attachments.DynamicChangeIndicationStop'>, <class 'mutwo.abjad_parameters.attachments.Fermata'>, <class 'mutwo.abjad_parameters.attachments.Glissando'>, <class 'mutwo.abjad_parameters.attachments.GraceNoteSequentialEvent'>, <class 'mutwo.abjad_parameters.attachments.Hairpin'>, <class 'mutwo.abjad_parameters.attachments.LaissezVibrer'>, <class 'mutwo.abjad_parameters.attachments.MarginMarkup'>, <class 'mutwo.abjad_parameters.attachments.Markup'>, <class 'mutwo.abjad_parameters.attachments.NaturalHarmonic'>, <class 'mutwo.abjad_parameters.attachments.Ornamentation'>, <class 'mutwo.abjad_parameters.attachments.Ottava'>, <class 'mutwo.abjad_parameters.attachments.Pedal'>, <class 'mutwo.abjad_parameters.attachments.Prall'>, <class 'mutwo.abjad_parameters.attachments.PreciseNaturalHarmonic'>, <class 'mutwo.abjad_parameters.attachments.RehearsalMark'>, <class 'mutwo.abjad_parameters.attachments.StringContactPoint'>, <class 'mutwo.abjad_parameters.attachments.Tempo'>, <class 'mutwo.abjad_parameters.attachments.Tie'>, <class 'mutwo.abjad_parameters.attachments.Tremolo'>, <class 'mutwo.abjad_parameters.attachments.Trill'>, <class 'mutwo.abjad_parameters.attachments.WoodwindFingering'>)

Default value for argument *abjad_attachment_classes* in `SequentialEventToAbjadVoiceConverter`.

mutwo.abjad_parameters

Table of content

Object	Documentation
mutwo.abjad_parameters.Arpeggio	
mutwo.abjad_parameters.Articulation	
mutwo.abjad_parameters.Trill	
mutwo.abjad_parameters.Cue	
mutwo.abjad_parameters.WoodwindFingering	
mutwo.abjad_parameters.Tremolo	
mutwo.abjad_parameters.ArtificalHarmonic	
mutwo.abjad_parameters.PreciseNaturalHarmonic	
mutwo.abjad_parameters.StringContactPoint	
mutwo.abjad_parameters.Pedal	
mutwo.abjad_parameters.Hairpin	
mutwo.abjad_parameters.BartokPizzicato	
mutwo.abjad_parameters.BreathMark	
mutwo.abjad_parameters.Fermata	
mutwo.abjad_parameters.NaturalHarmonic	
mutwo.abjad_parameters.Prall	
mutwo.abjad_parameters.Tie	
mutwo.abjad_parameters.DurationLineTriller	
mutwo.abjad_parameters.DurationLineDashed	
mutwo.abjad_parameters.Glissando	

continues on next page

Table 1 – continued from previous page

Object	Documentation
mutwo.abjad_parameters.BendAfter	
mutwo.abjad_parameters.LaissezVibrer	
mutwo.abjad_parameters.BarLine	
mutwo.abjad_parameters.Clef	
mutwo.abjad_parameters.Ottava	
mutwo.abjad_parameters.Markup	
mutwo.abjad_parameters.RehearsalMark	
mutwo.abjad_parameters.MarginMarkup	
mutwo.abjad_parameters.Ornamentation	
mutwo.abjad_parameters.Dynamic	Dynamic(dynamic_indicator: str = 'mf')
mutwo.abjad_parameters.Tempo	Tempo(reference_duration: Optional[tuple[int, int]] = (1, 4), units_per_minute: Union[int, tuple[int, int], NoneType] = 60, textual_indication: Optional[str] = None, dynamic_change_indication: Optional[str] = None, stop_dynamic_change_indicaton: bool = False, print_metronome_mark: bool = True)
mutwo.abjad_parameters.DynamicChangeIndicationStop	
mutwo.abjad_parameters.GraceNoteSequentialEvent	
mutwo.abjad_parameters.AfterGraceNoteSequentialEvent	

class Arpeggio(*direction=None*)

Bases: *Arpeggio*, *BangFirstAttachment*

Parameters
direction (*Optional[Literal['up', 'down']]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Articulation(*name=None*)

Bases: *Articulation*, *BangEachAttachment*

Parameters
name (*Optional[Literal['accent', 'marcato', 'staccatissimo', 'espressivo', 'staccato', 'tenuto', 'portato', 'upbow', 'downbow', 'flageolet', 'thumb', 'lheel', 'rheel', 'ltoe', 'rtoe', 'open', 'halfopen', 'snappizzicato', 'stopped', 'turn', 'reverseturn', 'trill', 'prall', 'mordent', 'prallprall', 'prallmordent', 'upprall', 'downprall', 'upmordent', 'downmordent', 'pralldown', 'prallup', 'lineprall', 'signumcongruentiae', 'shortfermata', 'fermata', 'longfermata', 'verylongfermata', 'segno', 'coda', 'varcoda', '^', '+', '-', '|', '>', '.', '_']]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Trill(*pitch=None*)

Bases: *Trill*, *BangFirstAttachment*

Parameters
pitch (*Optional[Pitch]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Cue(*cue_count=None*)

Bases: *Cue*, *BangFirstAttachment*

Parameters
`cue_count` (*Optional[int]*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class WoodwindFingering(*cc=None, left_hand=None, right_hand=None, instrument='clarinet'*)

Bases: *WoodwindFingering*, *BangFirstAttachment*

Parameters

- `cc` (*Optional[Tuple[str, ...]]*) –
- `left_hand` (*Optional[Tuple[str, ...]]*) –
- `right_hand` (*Optional[Tuple[str, ...]]*) –
- `instrument` (*str*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

fingering_size = 0.7

class Tremolo(*n_flags=None*)

Bases: *Tremolo*, *BangEachAttachment*

Parameters
`n_flags` (*Optional[int]*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class ArtificalHarmonic(*n_semitones=None*)

Bases: *ArtificalHarmonic*, *BangEachAttachment*

Parameters
`n_semitones` (*Optional[int]*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class PreciseNaturalHarmonic(*string_pitch=None, played_pitch=None, harmonic_note_head_style=True, parenthesize_lower_note_head=False*)

Bases: *PreciseNaturalHarmonic*, *BangEachAttachment*

Parameters

- `string_pitch` (*Optional[WesternPitch]*) –
- `played_pitch` (*Optional[WesternPitch]*) –
- `harmonic_note_head_style` (*bool*) –
- `parenthesize_lower_note_head` (*bool*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class StringContactPoint`(**args*, ***kwargs*)

Bases: *StringContactPoint*, *ToggleAttachment*

`process_leaf`(*leaf*, *previous_attachment*)

Parameters

- `leaf` (*Leaf*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
tuple[abjad.score.Leaf, ...]

`class Pedal`(*pedal_type=None*, *pedal_activity=True*)

Bases: *Pedal*, *ToggleAttachment*

Parameters

- `pedal_type` (*Optional[Literal['sustain', 'sostenuto', 'corda']]*) –
- `pedal_activity` (*Optional[bool]*) –

`process_leaf`(*leaf*, *previous_attachment*)

Parameters

- `leaf` (*Leaf*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
tuple[abjad.score.Leaf, ...]

`class Hairpin`(*symbol=None*, *niente=False*)

Bases: *Hairpin*, *ToggleAttachment*

Parameters

- `symbol` (*Optional[Literal['<', '>', '<>', '!']]*) –
- `niente` (*bool*) –

`process_leaf`(*leaf*, *_*)

Parameters

- `leaf` (*Leaf*) –
- `_` (*Optional[AbjadAttachment]*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
tuple[abjad.score.Leaf, ...]

`niente_literal = LilyPondLiteral('\\once \\override Hairpin.circled-tip = ##t', format_slot='opening')`

`class BartokPizzicato`(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangFirstAttachment*

Parameters
`is_active` (*bool*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class BreathMark`(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangFirstAttachment*

Parameters
`is_active` (*bool*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class Fermata`(*fermata_type=None*)

Bases: *Fermata*, *BangFirstAttachment*

Parameters
`fermata_type` (*Optional[Literal['shortfermata', 'fermata', 'longfermata', 'verylongfermata']]*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class NaturalHarmonic`(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangFirstAttachment*

Parameters
`is_active` (*bool*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Prall(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangFirstAttachment*

Parameters
`is_active` (*bool*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Tie(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangLastAttachment*

Parameters
`is_active` (*bool*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class DurationLineTriller(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangEachAttachment*

Parameters
`is_active` (*bool*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class DurationLineDashed(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangEachAttachment*

Parameters
`is_active` (*bool*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Glissando(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangLastAttachment*

Parameters
`is_active` (*bool*) –

process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`minimum_length = 5`

`thickness = 3`

class BendAfter(*bend_amount=None, minimum_length=3, thickness=3*)

Bases: *BendAfter*, *BangLastAttachment*

Parameters

- **bend_amount** (*Optional[float]*) –
- **minimum_length** (*Optional[float]*) –
- **thickness** (*Optional[float]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class LaissezVibrer(*is_active=False*)

Bases: *ExplicitPlayingIndicator*, *BangLastAttachment*

Parameters
is_active (*bool*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class BarLine(*abbreviation=None*)

Bases: *BarLine*, *BangLastAttachment*

Parameters
abbreviation (*Optional[str]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Clef(*name=None*)

Bases: *Clef*, *BangFirstAttachment*

Parameters
name (*Optional[str]*) –

process_leaf(*leaf*)

Parameters
leaf (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Ottava(*n_octaves=0*)

Bases: *Ottava*, *ToggleAttachment*

Parameters
n_octaves (*Optional[int]*) –

process_leaf(*leaf, previous_attachment*)

Parameters

- **leaf** (*Leaf*) –
- **previous_attachment** (*Optional[AbjadAttachment]*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[*`AbjadAttachment`*]*) –

Return type
tuple[abjad.score.Leaf, ...]

`class Markup`(*content=None*, *direction=None*)

Bases: *Markup*, *BangFirstAttachment*

Parameters

- `content` (*Optional[str]*) –
- `direction` (*Optional[str]*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class RehearsalMark`(*markup=None*)

Bases: *RehearsalMark*, *BangFirstAttachment*

Parameters
`markup` (*Optional[str]*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class MarginMarkup`(*content=None*, *context='Staff'*)

Bases: *MarginMarkup*, *BangFirstAttachment*

Parameters

- `content` (*Optional[str]*) –
- `context` (*Optional[str]*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class Ornamentation`(*direction=None*, *n_times=1*)

Bases: *Ornamentation*, *BangFirstAttachment*

Parameters

- `direction` (*Optional[Literal['up', 'down']]*) –
- `n_times` (*int*) –

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class Dynamic(*dynamic_indicator: str = 'mf'*)

Bases: *ToggleAttachment*

Parameters
: **dynamic_indicator** (*str*) –

classmethod from_indicator_collection(*indicator_collection*)

Always return None.

Dynamic can't be initialised from IndicatorCollection.

Parameters
: **indicator_collection** (IndicatorCollection) –

Return type
: *Optional*[AbjadAttachment]

process_leaf(*leaf, previous_attachment*)

Parameters
- **leaf** (*Leaf*) –
- **previous_attachment** (*Optional[AbjadAttachment]*) –

Return type
: *Union*[*Leaf*, *Sequence*[*Leaf*]]

dynamic_indicator: str = 'mf'

property is_active: bool

class Tempo(*reference_duration: Optional[tuple[int, int]] = (1, 4), units_per_minute: Union[int, tuple[int, int], NoneType] = 60, textual_indication: Optional[str] = None, dynamic_change_indication: Optional[str] = None, stop_dynamic_change_indicaton: bool = False, print_metronome_mark: bool = True*)

Bases: *BangFirstAttachment*

Parameters
- **reference_duration** (*Optional[tuple[int, int]]*) –
- **units_per_minute** (*Optional[Union[int, tuple[int, int]]]*) –
- **textual_indication** (*Optional[str]*) –
- **dynamic_change_indication** (*Optional[str]*) –
- **stop_dynamic_change_indicaton** (*bool*) –
- **print_metronome_mark** (*bool*) –

classmethod from_indicator_collection(*indicator_collection*)

Always return None.

Tempo can't be initialised from IndicatorCollection.

Parameters
: **indicator_collection** (IndicatorCollection) –

Return type
: *Optional*[AbjadAttachment]

process_leaf(*leaf*)

Parameters
: **leaf** (*Leaf*) –

Return type
: *Union*[*Leaf*, *Sequence*[*Leaf*]]

dynamic_change_indication: Optional[str] = None

property is_active: bool

print_metronome_mark: bool = True

`reference_duration: Optional[tuple[int, int]] = (1, 4)`

`stop_dynamic_change_indicaton: bool = False`

`textual_indication: Optional[str] = None`

`units_per_minute: Optional[Union[int, tuple[int, int]]] = 60`

`class DynamicChangeIndicationStop`

Bases: *BangFirstAttachment*

`classmethod from_indicator_collection`(*indicator_collection*)

Always return None.

DynamicChangeIndicationStop can't be initialised from IndicatorCollection.

Parameters
`indicator_collection` (IndicatorCollection) –

Return type
Optional[AbjadAttachment]

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`property is_active: bool`

`class GraceNoteSequentialEvent`(*grace_note_sequential_event*)

Bases: *BangFirstAttachment*

Parameters
`grace_note_sequential_event` (*BeforeGraceContainer*) –

`classmethod from_indicator_collection`(*indicator_collection*)

Always return None.

GraceNoteSequentialEvent can't be initialised from IndicatorCollection.

Parameters
`indicator_collection` (IndicatorCollection) –

Return type
Optional[AbjadAttachment]

`process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`property is_active: bool`

`class AfterGraceNoteSequentialEvent`(*after_grace_note_sequential_event*)

Bases: *BangLastAttachment*

Parameters
`after_grace_note_sequential_event` (*AfterGraceContainer*) –

`classmethod from_indicator_collection`(*indicator_collection*)

Always return None.

AfterGraceNoteSequentialEvent can't be initialised from IndicatorCollection.

Parameters
`indicator_collection` (IndicatorCollection) –

Return type
Optional[AbjadAttachment]

`process_leaf(leaf)`

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`property is_active: bool`

mutwo.abjad_parameters.abc

`class AbjadAttachment`

Bases: `ABC`

Abstract base class for all Abjad attachments.

`classmethod from_indicator_collection(indicator_collection)`

Initialize *`AbjadAttachment`* from `IndicatorCollection`.

If no suitable `Indicator` could be found in the collection the method will simply return None.

Parameters
`indicator_collection` (`IndicatorCollection`) –

Return type
Optional[AbjadAttachment]

`classmethod get_class_name()`

`abstract process_leaf_tuple(leaf_tuple, previous_attachment)`

Parameters

- `leaf_tuple` (*`tuple[abjad.score.Leaf, ...]`*) –
- `previous_attachment` (*`Optional[`*`AbjadAttachment`*`]`*) –

Return type
tuple[abjad.score.Leaf, ...]

`abstract property is_active: bool`

`class BangAttachment`

Bases: *`AbjadAttachment`*

Abstract base class for Abjad attachments which behave like a bang.

In Western notation one can differentiate between elements which only get notated if they change (for instance dynamics, tempo) and elements which have to be notated again and again to be effective (for instance arpeggi or tremolo). Attachments that inherit from *`BangAttachment`* represent elements which have to be notated again and again to be effective.

`abstract process_central_leaf(leaf)`

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

`abstract process_first_leaf(leaf)`

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

`abstract process_last_leaf(leaf)`

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
tuple[abjad.score.Leaf, ...]

`class BangEachAttachment`

Bases: *BangAttachment*

`process_central_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_first_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_last_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`abstract process_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`class BangFirstAttachment`

Bases: *BangAttachment*

`process_central_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_first_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_last_leaf`(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

abstract process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

class BangLastAttachment

Bases: *BangAttachment*

process_central_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

process_first_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

process_last_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Leaf

abstract process_leaf(*leaf*)

Parameters
`leaf` (*Leaf*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

process_leaf_tuple(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
tuple[abjad.score.Leaf, ...]

class ToggleAttachment

Bases: *AbjadAttachment*

Abstract base class for Abjad attachments which behave like a toggle.

In Western notation one can differentiate between elements which only get notated if they change (for instance dynamics, tempo) and elements which have to be notated again and again (for instance arpeggi or tremolo). Attachments that inherit from *ToggleAttachment* represent elements which only get notated if their value changes.

abstract process_leaf(*leaf*, *previous_attachment*)

Parameters

- `leaf` (*Leaf*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type
Union[*Leaf*, *Sequence*[*Leaf*]]

`process_leaf_tuple`(*leaf_tuple*, *previous_attachment*)

Parameters

- `leaf_tuple` (*tuple[abjad.score.Leaf, ...]*) –
- `previous_attachment` (*Optional[AbjadAttachment]*) –

Return type

tuple[abjad.score.Leaf, ...]

mutwo.abjad_parameters.configurations

Configure `mutwo.abjad_parameters

`CUSTOM_STRING_CONTACT_POINT_DICT = {'col legno tratto': 'c.l.t.'}`

Extends the predefined string contact points from `abjad.StringContactPoint`.

The `dict` has the form *{string_contact_point: abbreviation}*. It is used in the class *StringContactPoint*. You can override or update the default value of the variable to insert your own custom string contact points:

```
>>> from mutwo import abjad_parameters
>>> abjad_parameters.configurations.CUSTOM_STRING_CONTACT_POINT_DICT.update({"ebow": "eb"})
```

mutwo.abjad_parameters.constants

Constants to be used in `mutwo.abjad_parameters

`INDICATORS_TO_DETACH_FROM_MAIN_LEAF_AT_GRACE_NOTES_TUPLE = (<class 'abjad.indicators.TimeSignature.TimeSignature'>,)`

This is used in `mutwo.abjad_parameters.GraceNotes`.

Some indicators have to be detached from the main note and added to the first grace note, otherwise the resulting notation will first print the grace notes and afterwards the indicator (which is ugly and looks buggy).

mutwo.abjad_version

Table of content

`VERSION = '0.11.1'`

The version of the package `mutwo.abjad`.

mutwo.common_generators

Table of content

Object	Documentation
mutwo.common_generators.random_walk_noise	Generate an instance of Brownian motion (i.e. the Wiener process).
mutwo.common_generators.make_bruns_euclidean_algorithm_generator	Make generator which runs Bruns adaption of the Euclidean algorithm.
mutwo.common_generators.NonTerminal	Can be used as a Mixin to define context-free grammar.
mutwo.common_generators.Terminal	Can be used as a Mixin to define context-free grammar.
mutwo.common_generators.ContextFreeGrammarRule	Describe a context_free_grammar_rule for a *ContextFreeGrammar*
mutwo.common_generators.ContextFreeGrammar	Describe a context-free grammar and resolve non-terminals
mutwo.common_generators.ActivityLevel	Python implementation of Michael Edwards activity level algorithm.
mutwo.common_generators.reflected_binary_code	Make gray code where each tuple has *length* items with *modulus* different numbers.
mutwo.common_generators.Tendency	Tendency offers an interface for dynamically changing minima / maxima areas.
mutwo.common_generators.Backtracking	Abstract base class to implement a backtracking algorithm
mutwo.common_generators.IndexBasedBacktracking	Abstract base class for index based backtracking algorithms
mutwo.common_generators.euclidean	Return euclidean rhythm as described in a 2005 paper by G. T. Toussaint.
mutwo.common_generators.paradiddle	Generates rhythm using the paradiddle method described by G. T. Toussaint.
mutwo.common_generators.alternating_hands	Generates rhythm using the alternating hands method described by G. T. Toussaint.

random_walk_noise(*x0*, *n*, *dt*, *delta*, *out=None*, *random_state=None*)

Generate an instance of Brownian motion (i.e. the Wiener process).

Parameters

- **x0** (*float*) – the initial condition(s) (i.e. position(s)) of the Brownian motion.
- **n** (*int*) – the number of steps to take
- **dt** (*float*) – the time step
- **delta** (*float*) – delta determines the "speed" of the Brownian motion. The random variable of the position at time t, X(t), has a normal distribution whose mean is the position at time t=0 and whose variance is delta**2*t.
- **out** (*Optional[array]*) – If *out* is not None, it specifies the array in which to put the result. If *out* is None, a new numpy array is created and returned.
- **random_state** (*Optional[int]*) – set the random seed of the pseudo-random generator.

Returns

A numpy array of floats with shape *x0.shape + (n,)*.

Return type

array

X(t) = X(0) + N(0, delta**2 * t; 0, t)

where N(a,b; t0, t1) is a normally distributed random variable with mean a and variance b. The parameters t0 and t1 make explicit the statistical independence of N on different time intervals; that is, if [t0, t1) and [t2, t3) are disjoint intervals, then N(a, b; t0, t1) and N(a, b; t2, t3) are independent.

Written as an iteration scheme,

X(t + dt) = X(t) + N(0, delta**2 * dt; t, t+dt)

If *x0* is an array (or array-like), each value in *x0* is treated as an initial condition, and the value returned is a numpy array with one more dimension than *x0*.

Note that the initial value *x0* is not included in the returned array.

This code has been copied from the scipy cookbook:

https://scipy-cookbook.readthedocs.io/items/BrownianMotion.html

make_bruns_euclidean_algorithm_generator(*element_tuple*, *matrix=array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *subtraction_index=1*)

Make generator which runs Bruns adaption of the Euclidean algorithm.

Parameters

- **element_tuple** (*tuple[_BrunEuclideanElement, _BrunEuclideanElement, _BrunEuclideanElement]*) – The initial elements which gets re-calculated after each step. Type doesn't matter; objects only need to have the following magic methods: __sub__, __lt__ and __gt__.

- **matrix** (*np.array*) – The initial matrix.
- **subtraction_index** (*Literal[1, 2]*) – This parameter has been added for the adaption of the function in `make_wilsons_brun_euclidean_algorithm_generator()` and is not part of Bruns original algorithm. It describes whether in each step the first element gets subtracted by the second (original) or by the third (Wilson adaption) element.

Return type
Generator

This algorithm has been described by V. Brun in his paper "EUCLIDEAN ALGORITHMS AND MUSICAL THEORY" (1964).

Example:

```
>>> import fractions
>>> from mutwo.generators import brun
>>> bruns_euclidean_algorithm_generator = brun.make_bruns_euclidean_algorithm_generator(
>>>     (
>>>         fractions.Fraction(2, 1),
>>>         fractions.Fraction(3, 2),
>>>         fractions.Fraction(5, 4),
>>>     )
>>> )
>>> next(bruns_euclidean_algorithm_generator)
```

reflected_binary_code(*length*, *modulus*)

Make gray code where each tuple has *length* items with *modulus* different numbers.

Parameters

- **length** (*int*) – how long one code is
- **modulus** (*int*) – how many different numbers are included

Return type
tuple[tuple[int, ...], ...]

Example:

```
>>> from mutwo.generators import gray
>>> gray.reflected_binary_code(2, 2)
((0, 0), (0, 1), (1, 1), (1, 0))
>>> gray.reflected_binary_code(3, 2)
((0, 0, 0),
(0, 0, 1),
(0, 1, 1),
(0, 1, 0),
(1, 1, 0),
(1, 1, 1),
(1, 0, 1),
(1, 0, 0))
>>> gray.reflected_binary_code(2, 3)
((0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (1, 0), (2, 0), (2, 1), (2, 2))
```

Basic code has been copied from:
https://yetalengthothermodulusathblog.com/tag/gray-codes/

euclidean(*size*, *distribution*)

Return euclidean rhythm as described in a 2005 paper by G. T. Toussaint.

Parameters

- **size** (*int*) – how many beats the rhythm contains
- **distribution** (*int*) – how many beats are played

Returns
The rhythm in relative time.

Return type
tuple[int, ...]

Example:

```
>>> from mutwo.generators import toussaint
>>> toussaint.euclidean(8, 4)
(2, 2, 2, 2)
>>> toussaint.euclidean(7, 5)
(2, 1, 1, 2, 1)
```

The title of Toussaints paper is "The Euclidean Algorithm Generates Traditional Musical Rhythms".

`paradiddle`(*size*)

Generates rhythm using the paradiddle method described by G. T. Toussaint.

Parameters

`size` (*int*) – how many beats the resulting rhythm shall last. 'Size' has to be divisible by 2 because of the symmetrical structure of the generated rhythm.

Returns

Return nested tuple that contains two tuple where each tuple represents one rhythm (both rhythms are complementary to each other). The rhythms are encoded in absolute time values.

Return type

tuple[tuple[int, ...], ...]

Example:

```
>>> from mutwo.generators import toussaint
>>> toussaint.paradiddle(8)
((0, 2, 3, 5), (1, 4, 6, 7))
>>> toussaint.paradiddle(6)
((0, 4, 5), (1, 2, 3))
```

The paradiddle algorithm has been described by Godfried T. Toussaint in his paper 'Generating "Good" Musical Rhythms Algorithmically'.

`alternating_hands`(*seed_rhythm*)

Generates rhythm using the alternating hands method described by G. T. Toussaint.

Parameters

`seed_rhythm` (*tuple[int, ...]*) – rhythm that shall be distributed on two hands.

Returns

Return nested tuple that contains two tuple where each tuple represents one rhythm (both rhythms are complementary to each other). The rhythms are encoded in absolute time values.

Return type

tuple[tuple[int, ...], ...]

Example:

```
>>> from mutwo.generators import toussaint
>>> toussaint.alternating_hands((2, 2))
((0, 6), (2, 4))
>>> toussaint.alternating_hands((3, 2, 2))
((0, 5, 10), (3, 7, 12))
```

The alternating hands algorithm has been described by Godfried T. Toussaint in his paper 'Generating "Good" Musical Rhythms Algorithmically'.

`class NonTerminal`

Bases: `object`

Can be used as a Mixin to define context-free grammar.

`class Terminal`

Bases: `object`

Can be used as a Mixin to define context-free grammar.

`class ContextFreeGrammarRule`(*left_side*, *right_side*)

Bases: `object`

Describe a context_free_grammar_rule for a *ContextFreeGrammar*

Parameters

- `left_side` (`NonTerminal`) –
- `right_side` (*`tuple[Union[`*`mutwo.common_generators.chomksy.NonTerminal`*`,`*`mutwo.common_generators.chomksy.Terminal`*`], ...]`*) –

`left_side:` *`NonTerminal`*

`right_side: tuple[Union[`*`mutwo.common_generators.chomksy.NonTerminal`*`,` *`mutwo.common_generators.chomksy.Terminal`*`], ...]`

`class ContextFreeGrammar`(*context_free_grammar_rule_sequence*)

Bases: `object`

Describe a context-free grammar and resolve non-terminals

Parameters

`context_free_grammar_rule_sequence` (*`Sequence[`*`ContextFreeGrammarRule`*`]`*) – A sequence of *`ContextFreeGrammarRule`* objects. It is allowed to provide multiple context_free_grammar_rules with the same **:attribute:`left_side`**.

This is a very reduced implementation of a context-free grammar which only provides the most basic functions. It is not made for the purpose of parsing text but rather as a technique to generate algorithmic data (for the sake of art creation). Therefore it is all about the resolution of start objects to variants of this start.

`get_context_free_grammar_rule_tuple`(*non_terminal*)

Find all defined context_free_grammar_rules for the provided *`NonTerminal`*.

Parameters

`non_terminal` (`NonTerminal`) – The left side element of the *`ContextFreeGrammarRule`*.

Return type

tuple[*mutwo.common_generators.chomksy.ContextFreeGrammarRule*, ...]

`resolve`(*start*, *limit=None*)

Resolve until only *`Terminal`* are left or the limit is reached.

Parameters

- `start` (`NonTerminal`) – The start value.
- `limit` (*`Optional[int]`*) – The maximum node levels until the function returns a tree. If it is set to *None* it will only stop once all nodes are *`Terminal`*.

Return type

Tree

`resolve_one_layer`(*tree*)

Resolve all leaves of the tree.

Parameters

`tree` (*`treelib.Tree`*) – The tree from which all leaves should be resolved.

Returns

True if any leaf has been resolved and *False* if no resolution has happened (e.g. if there are only *`Terminal`* left).

Return type

bool

`property context_free_grammar_rule_tuple: tuple[`*`mutwo.common_generators.chomksy.ContextFreeGrammarRule`*`, ...]`

Get all defined rules

`property non_terminal_tuple: tuple[`*`mutwo.common_generators.chomksy.NonTerminal`*`, ...]`

`property terminal_tuple: tuple[`*`mutwo.common_generators.chomksy.Terminal`*`, ...]`

`class ActivityLevel`(*start_at=0*)

Bases: `object`

Python implementation of Michael Edwards activity level algorithm.

Parameters

`start_at` (*`int`*) – from which pattern per level shall be started (can be either 0, 1 or 2)

Activity Levels is a concept derived from Michael Edwards. Quoting Michael Edwards, Activity Levels are an "object for determining (deterministically) on a call-by-call basis whether a process is active or not (boolean). This is determined by nine 10-element lists (actually three versions of each) of hand-coded 1s and 0s, each list representing an 'activity-level' (how active the process should be). The first three 10-element lists have only one 1 in them, the rest being zeros. The second three have two 1s, etc. Activity-levels of 0 and 10 would return never active and always active respectively.".

Example:

```
>>> from mutwo.generators import edwards
>>> activity_levels = edwards.ActivityLevel()
>>> activity_levels(0)  # activity level 0 will always return False
False
>>> activity_levels(10)  # activity level 10 will always return True
True
>>> activity_levels(7)  # activity level 7 will mostly return True
True
>>> tuple(activity_levels(7) for _ in range(10))
(True, False, True, True, False, True, True, False, True, True)
```

`class Tendency`(*minima_curve*, *maxima_curve*, *random_seed=100*)

Bases: `object`

Tendency offers an interface for dynamically changing minima / maxima areas.

Parameters

- **`minima_curve`** (`core_events.Envelope`) – The curve which describes the smallest allowed value over the time axis.
- **`maxima_curve`** (`core_events.Envelope`) – The curve which describes the biggest allowed value over the time axis.
- **`random_seed`** (*`int`*) – The random seed which shall be set.

The class is based on Gottfried Michael Koenigs algorithm of "Tendenz-Masken" in his program "Projekt 2" where those minima / maxima areas represent probability fields.

Example:

```
>>> import core_events
>>> from mutwo.generators import koenig
>>> minima_curve = core_events.Envelope.from_points((0, 0), (1, 1), (2, 0))
>>> maxima_curve = core_events.Envelope.from_points((0, 1), (1, 2), (2, 3))
>>> my_tendency = koenig.Tendency(minima_curve, maxima_curve)
>>> my_tendency.value_at(0.5)
0.6456692551041303
>>> my_tendency.value_at(0.5)
0.9549270045140213
```

`range_at`(*time*)

Get minima / maxima range at requested time.

Parameters
`time` (*`float`*) –

Return type
Range

`value_at`(*time*)

Get value at requested time.

Parameters
`time` (*`float`*) –

Return type
float

`property maxima_curve:` *`Envelope`*

`property minima_curve:` *`Envelope`*

`class Backtracking`

Bases: `ABC`

Abstract base class to implement a backtracking algorithm

By inheriting from this class, various backtracking algorithms can be implemented. In order to do so the user has to override a set of abstract methods. The abstract methods include:

- **:abstractmethod:`Backtracking.is_valid`**
- **:abstractmethod:`Backtracking.solution_count`**
- **:abstractmethod:`Backtracking.append_new_element`**
- **:abstractmethod:`Backtracking.update_last_element`**
- **:abstractmethod:`Backtracking.can_last_element_be_updated`**

Furthermore it may be helpful to override the following method (even though there is a valid working implementation):

- **:method:`Backtracking.element_list_to_solution`**

Please see the methods documentation for more details.

The implementation of this backtracking algorithm makes a distinction between an element list and a solution. A solution is created by an element list. A solution is the output a user wants to get, but an element list is an object which is used internally in order to solve the problem. When implementing a backtracking algorithm by using this interface the user doesn't have to make the distinction between both (and in this case treat both in the same way).

The most common use case for this distinction is by having a set of items which can appear in the solution and a list of indices which item of set shall be used. In this case the element_list is actually a list of indices. This use case is implemented in the *`IndexBasedBacktracking`* class.

Bitner and Reingold [2] credit Derrick H. Lehmer with first using the term 'backtrack' in the 1950s..

`abstract append_new_element`(*element_list*)

Append new element to element list.

Parameters
`element_list` (*`list[Any]`*) – The element list to which a new element shall be appended.

`abstract can_last_element_be_updated`(*element_list*)

Checks if the last element of the list can be incremented.

Parameters
`element_list` (*`list[Any]`*) – The element list which last value shall be checked.

Return type
bool

`element_list_to_solution`(*element_list*)

Converts an element list to the final solution

Parameters
`element_list` (*`list[Any]`*) – The element list to be converted.

Return type
tuple[*Any*, ...]

`abstract is_valid`(*element_list*)

Checks if an element list provides an acceptable solution.

Returns
True if the solution is acceptable and *False* if the solution is rejected.

Parameters
`element_list` (*`list[Any]`*) –

Return type
bool

`solve`(*return_element_list=False*)

Apply backtracking algorithm.

Parameters
`return_element_list` (*`bool`*) – If set to *True* the function will not only return the solution, but also the element list.

Return type
Union[tuple[*Any*, ...], tuple[tuple[*Any*, ...], list[*Any*]]]

`abstract update_last_element`(*element_list*)

Increments value of the last element in an element_list.

Parameters

`element_list` (*list[Any]*) – The element list which last value shall be updated.

This function should raise an Exception in case the last element can't be updated.

`abstract property solution_count: int`

Return expected solution size

`class IndexBasedBacktracking`

Bases: *Backtracking*

Abstract base class for index based backtracking algorithms

This class implements concrete solutions for the following methods which are inherited from the parent class *Backtracking*:

- **:abstractmethod:`Backtracking.append_new_element`**
- **:abstractmethod:`Backtracking.update_last_element`**
- **:abstractmethod:`Backtracking.can_last_element_be_updated`**

The following methods still have to be implemented:

- **:abstractmethod:`Backtracking.is_valid`**
- **:abstractmethod:`Backtracking.solution_count`**

(Please consult for more information the documentation of *Backtracking*).

Furthermore the class adds new abstract methods to be implemented by child classes:

- **:abstractmethod:`IndexBasedBacktracking.element_index_to_item_sequence`**

Example:

```
>>> import itertools
>>> from mutwo import common_generators
>>> class QueenProblem8(common_generators.IndexBasedBacktracking):
        point_list = list(itertools.combinations_with_replacement(range(queen_count), 2))
        point_list.extend(
            [tuple(reversed(point)) for point in point_list if len(set(point)) == 2]
        )
        def element_index_to_item_sequence(self, element_index, element_list):
            return self.point_list
        @property
        def solution_count(self):
            # 8 queens problem!
            return 8
        def is_valid(self, element_list):
            solution = self.element_list_to_solution(element_list)
            for queen0, queen1 in itertools.combinations(solution, 2):
                # x != x, y != y
                is_valid = all(value0 != value1 for value0, value1 in zip(queen0, queen1))
                difference_x, difference_y = (value0 - value1 for value0, value1 in zip(queen0, queen1))
                is_valid = is_valid and (difference_x != difference_y)
                if not is_valid: return False
            return True
>>> queen_problem_8 = QueenProblem8()
>>> queen_problem_8.solve()
```

`append_new_element`(*element_list*)

Append new element to element list.

Parameters

`element_list` (*list[Any]*) – The element list to which a new element shall be appended.

`can_last_element_be_updated`(*element_list*)

Checks if the last element of the list can be incremented.

Parameters

`element_list` (*list[Any]*) – The element list which last value shall be checked.

Return type

bool

`abstract element_index_to_item_sequence`(*element_index*, *element_list*)

Get a sequence of items to choose from for a specific element

Parameters

- `element_index` (*int*) – The index of the element for which a sequence of solutions shall be returned.
- `element_list` (*list[Any]*) – The current element list

Return type

Sequence[*Any*]

`element_list_to_solution`(*element_list*)

Converts an element list to the final solution

Parameters

`element_list` (*list[Any]*) – The element list to be converted.

Return type

tuple[*Any*, ...]

`update_last_element`(*element_list*)

Increments value of the last element in an element_list.

Parameters

`element_list` (*list[Any]*) – The element list which last value shall be updated.

This function should raise an Exception in case the last element can't be updated.

mutwo.common_generators.constants

Constants which are used in *mutwo.common_generators*.

```
ACTIVITY_LEVEL_TUPLE = (((0,), (0,), (0,)), ((1, 0, 0, 0, 0, 0, 0, 0, 0, 0), (0, 0, 0, 1, 0, 0, 0, 0, 0, 0), (0,
0, 0, 0, 0, 0, 1, 0, 0, 0)), ((1, 0, 0, 0, 0, 0, 1, 0, 0, 0), (0, 0, 0, 1, 0, 1, 0, 0, 0, 0), (0, 0, 0, 0, 0, 0,
1, 1, 0, 0)), ((1, 0, 0, 0, 1, 0, 1, 0, 0, 0), (0, 0, 0, 1, 0, 1, 1, 0, 0, 0), (0, 0, 1, 0, 0, 0, 1, 1, 0, 0)),
((1, 0, 0, 0, 1, 0, 1, 1, 0, 0), (0, 1, 0, 1, 0, 1, 1, 0, 0, 0), (0, 0, 1, 0, 0, 0, 1, 1, 0, 1)), ((1, 1, 0, 0,
1, 0, 1, 1, 0, 0), (0, 1, 0, 1, 0, 1, 1, 0, 0, 1), (0, 0, 1, 0, 1, 0, 1, 1, 0, 1)), ((1, 1, 0, 1, 1, 0, 1, 1, 0,
0), (0, 1, 0, 1, 0, 1, 1, 0, 1, 1), (0, 1, 1, 0, 1, 0, 1, 1, 0, 1)), ((1, 1, 0, 1, 1, 0, 1, 1, 0, 1), (1, 1, 0,
1, 0, 1, 1, 0, 1, 1), (1, 1, 1, 0, 1, 0, 1, 1, 0, 1)), ((1, 1, 0, 1, 1, 1, 1, 1, 0, 1), (1, 1, 1, 1, 0, 1, 1, 0,
1, 1), (1, 1, 1, 0, 1, 1, 1, 1, 0, 1)), ((1, 1, 0, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1, 1, 0, 1, 1, 1, 1, 1), (1, 1,
1, 1, 1, 1, 1, 1, 0, 1)), ((1,), (1,), (1,)))
```

Definition of activity level pattern. Pattern are copied from Michael Edwards Common Lisp composition software 'slippery-chicken'.

mutwo.common_utilities

Table of content

Object	Documentation
mutwo.common_utilities.InvalidMinimaCurveAndMaximaCurveCombination	Raise for invalid envelope combinations in *mutwo.common_generators.Tendency*.
mutwo.common_utilities.UnequalEnvelopeDurationError	
mutwo.common_utilities.InvalidStartAtValueError	Raise for invalid error of 'start_at' in *mutwo.common_generators.ActivityLevel*
mutwo.common_utilities.NoSolutionFoundError	Raise in case backtracking algorithm can't find any solution

`class InvalidMinimaCurveAndMaximaCurveCombination`

Bases: `Exception`

Raise for invalid envelope combinations in *`mutwo.common_generators.Tendency`*.

`class UnequalEnvelopeDurationError`(*minima_curve*, *maxima_curve*)

Bases: *`InvalidMinimaCurveAndMaximaCurveCombination`*

Parameters

- **`minima_curve`** (`Envelope`) –
- **`maxima_curve`** (`Envelope`) –

`class InvalidStartAtValueError`(*start_at*)

Bases: `ValueError`

Raise for invalid error of 'start_at' in *`mutwo.common_generators.ActivityLevel`*

Parameters

`start_at` (*`int`*) –

`class NoSolutionFoundError`

Bases: `Exception`

Raise in case backtracking algorithm can't find any solution

mutwo.common_version

Table of content

`VERSION = '0.9.1'`

The version of the package `mutwo.common`.

mutwo.core_constants

Table of content

Definition of global variables which are used all over mutwo.

`DurationType`

Type variable to arguments and return values for *duration*. This can be any real number (float, integer, fraction).

alias of `Union[float, Fraction, int]`

`ParameterType = typing.Any`

Type variable to assign to arguments and return values which expect objects from the `mutwo.core.parameters` module, but could actually be anything.

`Real`

The main reason for this constant is a mypy issue with Pythons buildin [numbers module](https://docs.python.org/3/library/numbers.html) which is documented [here](https://github.com/python/mypy/issues/3186). Mypy doesn't accept numbers abstract base classes. Until numbers will be supported users have to define their own typing data for general number classes. PEP 3141 recommends users to simply annotate arguments with 'float', but this wouldn't include *fractions.Fraction* which is often necessary in musical contexts (as github user arseniiv also remarked).

alias of `Union[float, Fraction, int]`

mutwo.core_converters

Table of content

Convert data from and to mutwo.

Object	Documentation
mutwo.core_converters.SimpleEventToAttribute	Extract from a simple event an attribute.
mutwo.core_converters.MutwoParameterDictToKeywordArgument	Extract from a dict of mutwo parameters specific objects.
mutwo.core_converters.MutwoParameterDictToDuration	Extract from a dict of mutwo parameters the duration.
mutwo.core_converters.MutwoParameterDictToSimpleEvent	Convert a dict of mutwo parameters to a *mutwo.core_events.SimpleEvent*
mutwo.core_converters.UnknownObjectToObject	Helper to simplify standardisation of syntactic sugar.
mutwo.core_converters.TempoPointConverter	Convert a `TempoPoint` with BPM to beat-length-in-seconds.
mutwo.core_converters.TempoConverter	Apply tempo curves on mutwo events
mutwo.core_converters.EventToMetrizedEvent	Apply tempo envelope of event on itself

`class` **`SimpleEventToAttribute`**(*attribute_name*, *exception_value*)

Bases: *Converter*

Extract from a simple event an attribute.

Parameters

- **attribute_name** (*str*) – The name of the attribute which is fetched from a *mutwo.core_events.SimpleEvent*.
- **exception_value** (*Any*) – This value is returned in case an *AttributeError* raises .

`convert`(*simple_event_to_convert*)

Extract from a *mutwo.core_events.SimpleEvent* an attribute.

Parameters

simple_event_to_convert (mutwo.core_events.SimpleEvent) – The *mutwo.core_events.SimpleEvent* from which an attribute shall be extracted.

Return type

Any

Example:

```
>>> from mutwo import core_converters
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(duration=10)
>>> simple_event_to_duration = core_converters.SimpleEventToAttribute(
        'duration', 0
    )
>>> simple_event_to_duration.convert(simple_event)
10
>>> simple_event_to_pasta = core_converters.SimpleEventToAttribute(
        'pasta', 'spaghetti'
    )
>>> simple_event_to_pasta.convert(simple_event)
'spaghetti'
>>> simple_event.pasta = 'tagliatelle'
>>> simple_event_to_pasta.convert(simple_event)
'tagliatelle'
```

`class` **`MutwoParameterDictToKeywordArgument`**(*mutwo_parameter_to_search_name*, *keyword=None*)

Bases: *Converter*

Extract from a dict of mutwo parameters specific objects.

Parameters

- **`mutwo_parameter_to_search_name`** (*str*) – The parameter name which should be fetched from the MutwoParameterDict (if it exists).
- **`keyword`** (*Optional[str]*) – The keyword string to return. If no argument is given it will use the same value as **:param:`mutwo_parameter_to_search_name`**.

Example:

```
>>> from mutwo import core_converters
>>> from mutwo import music_parameters
>>> mutwo_parameter_dict_to_keyword_argument = core_converters.MutwoParameterDictToKeywordArgument('pitch')
>>> mutwo_parameter_dict_to_keyword_argument.convert(
    {'pitch': music_parameters.WesternPitch('c')}
)
('pitch', music_parameters.WesternPitch(c4))
```

`convert`(*mutwo_parameter_dict_to_convert*)

Parameters
`mutwo_parameter_dict_to_convert` (*dict[str, Any]*) –

Return type
Optional[tuple[str, *Any*]]

class `MutwoParameterDictToDuration`(*duration_to_search_name=None*, *duration_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Extract from a dict of mutwo parameters the duration.

Parameters

- **`duration_to_search_name`** (*Optional[str]*) – The name of the duration which shall be searched for in the MutwoParameterDict. If *None* the value of the global constants *mutwo.core_converters.configurations.DEFAULT_DURATION_TO_SEARCH_NAME* will be used. Default to *None*.
- **`duration_keyword_name`** (typing.Optional[str] *mutwo.core_converters.configurations.DEFAULT_DURATION_KEYWORD_NAME*.) – The name of the duration keyword for the event. If *None* the value of the global constants *mutwo.core_converters.configurations.DEFAULT_DURATION_KEYWORD_NAME* will be used. Default to *None*.

class `MutwoParameterDictToSimpleEvent`(*mutwo_parameter_dict_to_keyword_argument_sequence=None*, *simple_event_class=<class 'mutwo.core_events.basic.SimpleEvent'>*)

Bases: *Converter*

Convert a dict of mutwo parameters to a *mutwo.core_events.SimpleEvent*

Parameters

- **`mutwo_parameter_dict_to_keyword_argument_sequence`**(*Optional[Sequence[*MutwoParameterDictToKeywordArgu
  – A sequence of *MutwoParameterDictToKeywordArgument*. If set to *None* a sequence with *MutwoParameterDictToDuration* will be created. Default to *None*.
- **`simple_event_class`** (*Type[*core_events.SimpleEvent*]*) – Default to *mutwo.core_events.SimpleEvent*.

`convert`(*mutwo_parameter_dict_to_convert*)

Parameters
`mutwo_parameter_dict_to_convert` (*dict[str, Any]*) –

Return type
SimpleEvent

class `UnknownObjectToObject`(*type_tuple_and_callable_tuple*)

Bases: *Converter*, Generic[T]

Helper to simplify standardisation of syntactic sugar.

Parameters

- **`type_tuple_to_callable_dict`** – Define which types are converted by which methods.

- **type_tuple_and_callable_tuple** (*tuple[tuple[Type, ...], Callable]*) –

Example:

```
>>> from mutwo impot core_converters
>>> anything_to_string = core_converters.UnknownObjectToObject[str](
>>>     (
>>>         ((float, int, list), str),
>>>         ((tuple,), lambda t: str(len(t))),
>>>         ([], lambda _: "..."),
>>>     )
>>> )
>>> anything_to_string.convert(100)
"100"
>>> anything_to_string.convert(7.32)
"7.32"
>>> anything_to_string.convert((1, 2, 3))
"3"
>>> anything_to_string.convert(b'')
"..."
```

`convert`(*unknown_object_to_convert*)

Parameters
`unknown_object_to_convert` (*Any*) –

Return type
T

`class TempoPointConverter`

Bases: *Converter*

Convert a `TempoPoint` with BPM to beat-length-in-seconds.

A *TempoPoint* is defined as an object that has a particular tempo in beats per seconds (BPM) and a reference value (1 for a quarter note, 4 for a whole note, etc.). Besides elaborate `mutwo.parameters.tempos.TempoPoint` objects, any number can also be interpreted as a *TempoPoint*. In this case the number simply represents the BPM number and the reference will be set to 1. The returned beat-length-in-seconds always indicates the length for one quarter note.

Example:

```
>>> from mutwo.converters import symmetrical
>>> tempo_point_converter = symmetrical.tempos.TempoPointConverter()
```

`convert`(*tempo_point_to_convert*)

Converts a *TempoPoint* to beat-length-in-seconds.

Parameters
`tempo_point_to_convert` (*Union[TempoPoint, float, Fraction, int]*) – A tempo point defines the active tempo from which the beat-length-in-seconds shall be calculated. The argument can either be any number (which will be interpreted as beats per minute [BPM]) or a `mutwo.parameters.tempos.TempoPoint` object.

Returns
The duration of one beat in seconds within the passed tempo.

Return type
float

Example:

```
>>> from mutwo.converters import symmetrical
>>> converter = symmetrical.tempos.TempoPointConverter()
>>> converter.convert(60)  # one beat in tempo 60 bpm takes 1 second
1
>>> converter.convert(120)  # one beat in tempo 120 bpm takes 0.5 second
0.5
```

`TempoPoint`

alias of `Union`[*TempoPoint*, `float`, `Fraction`, `int`] `class` **`TempoConverter`**(*tempo_envelope*, *apply_converter_on_events_tempo_envelope=True*)

Bases: *`EventConverter`*

Apply tempo curves on mutwo events

Parameters

- **`tempo_envelope`** (`TempoEnvelope`) – The tempo curve that shall be applied on the mutwo events. This is expected to be a `core_events.TempoEnvelope` which values are filled with numbers that will be interpreted as BPM [beats per minute]) or with *`mutwo.core_parameters.TempoPoint`* objects.
- **`apply_converter_on_events_tempo_envelope`** (*`bool`*) – If set to *True* the converter will also adjust the `tempo_envelope` attribute of each converted event. Default to *True*.

Example:

```
>>> from mutwo import core_converters
>>> from mutwo import core_events
>>> from mutwo import core_parameters
>>> tempo_envelope = core_events.Envelope(
>>>     [[0, tempos.TempoPoint(60)], [3, 60], [3, 30], [5, 50]],
>>> )
>>> my_tempo_converter = core_converters.TempoConverter(tempo_envelope)
```

`convert`(*event_to_convert*)

Apply tempo curve of the converter to the entered event.

The method doesn't change the original event, but returns a copied version with different values for its duration attributes depending on the tempo curve.

Parameters

`event_to_convert` (`Event`) – The event to convert. Can be any object that inherits from `mutwo.events.abc.Event`. If the event that shall be converted is longer than the tempo curve of the `TempoConverter`, then the last tempo of the curve will be hold.

Returns

A new `Event` object which duration property has been adapted by the tempo curve of the `TempoConverter`.

Return type

Event

Example:

```
>>> from mutwo import core_converters
>>> from mutwo import core_events
>>> from mutwo import core_parameters
>>> tempo_envelope = core_events.Envelope(
>>>     [[0, tempos.TempoPoint(60)], [3, 60], [3, 30], [5, 50]],
>>> )
>>> my_tempo_converter = core_converters.TempoConverter(tempo_envelope)
>>> my_events = core_events.SequentialEvent([core_events.SimpleEvent(d) for d in (3, 2, 5)])
>>> my_tempo_converter.convert(my_events)
SequentialEvent([SimpleEvent(duration = 3.0), SimpleEvent(duration = 1.5), SimpleEvent(duration = 2.
→5)])
```

`class` **`EventToMetrizedEvent`**(*skip_level_count=None*, *maxima_depth_count=None*)

Bases: *`SymmetricalEventConverter`*

Apply tempo envelope of event on itself

Parameters

- **`skip_level_count`** (*`Optional[int]`*) –
- **`maxima_depth_count`** (*`Optional[int]`*) –

`convert`(*event_to_convert*)

Apply tempo envelope of event on itself

Parameters

`event_to_convert` (`Event`) –

Return type
Event

mutwo.core_converters.abc

Defining the public API for any converter class.

`class Converter`

Bases: `ABC`

Abstract base class for all Converter classes.

Converter classes are defined as classes that convert data between two different encodings. Their only public method (besides initialisation) should be a *convert* method. The first argument of the convert method should be the data to convert.

`abstract convert`(*event_or_parameter_or_file_to_convert*, **args*, ***kwargs*)

Parameters
`event_or_parameter_or_file_to_convert` (*Any*) –

Return type
Any

`class EventConverter`

Bases: *Converter*

Abstract base class for Converter which handle mutwo events.

This class helps building new classes which convert mutwo events with few general private methods (and without adding any new public method). Converting mutwo event often involves the same pattern: due to the nested structure of an Event, the converter has to iterate through the different layers until it reaches leaves (any class that inherits from *mutwo.core_events.SimpleEvent*). This common iteration process and the different time treatment between *mutwo.core_events.SequentialEvent* and *mutwo.core_events.SimultaneousEvent* are implemented in *EventConverter*. For writing a new EventConverter class, one only has to override the abstract method `_convert_simple_event()` and the abstract method `convert()` (where one will perhaps call `_convert_event()`.).

Example:

The following example defines a dummy class for demonstrating how to use EventConverter.

```
>>> from mutwo import core_converters
>>> class DurationPrintConverter(core_converters.abc.EventConverter):
>>>     def _convert_simple_event(self, event_to_convert, absolute_entry_delay):
>>>         return "{}: {}".format(absolute_entry_delay, event_to_convert.duration),
>>>     def convert(self, event_to_convert):
>>>         data_per_event = self._convert_event(event_to_convert, 0)
>>>         [print(data) for data in data_per_event]
>>> # now test with random event
>>> import random
>>> from mutwo import core_events
>>> random.seed(100)
>>> random_event = core_events.SimultaneousEvent(
>>>     [
>>>        core_events.SequentialEvent(
>>>             [
>>>                 core_events.SimpleEvent(random.uniform(0.5, 2))
>>>                  for _ in range(random.randint(2, 5))
>>>             ]
>>>         )
>>>         for _ in range(random.randint(1, 3))
>>>     ]
>>> )
>>> DurationPrintConverter().convert(random_event)
0: 1.182390506771032
1.182390506771032: 1.6561757084885333
2.8385662152595654: 1.558269840401042
4.396836055660607: 1.5979384595498836
5.994774515210491: 1.1502716523431056
```

`class SymmetricalEventConverter`

Bases: *EventConverter*

Abstract base class for Converter which handle mutwo core_events.

This converter is a more specified version of the *EventConverter*. It helps for building converters which aim to return mutwo core_events.

mutwo.core_converters.configurations

Configure *mutwo.core_converters*

DEFAULT_DURATION_KEYWORD_NAME = 'duration'

Default value for `duration_keyword_name` parameter in *mutwo.core_converters.MutwoParameterDictToDuration*

DEFAULT_DURATION_TO_SEARCH_NAME = 'duration'

Default value for `duration_to_search_name` parameter in *mutwo.core_converters.MutwoParameterDictToDuration*

mutwo.core_events

Table of content

Time-based Event abstractions.

Event objects can be understood as the core objects of the *mutwo* framework. They all own a `duration` attribute (which can be any number). Further more complex Event classes with more relevant attributes can be generated through inheriting from basic classes. *mutwo* already offers support for several more complex representations (for instance *mutwo.music_events.NoteLike*). The most often used classes may be: - *mutwo.core_events.SimpleEvent* - *mutwo.core_events.SequentialEvent* - *mutwo.core_events.SimultaneousEvent*

Object	Documentation
mutwo.core_events.SimpleEvent	Event-Object which doesn't contain other Event-Objects (the node or leaf).
mutwo.core_events.SequentialEvent	Event-Object which contains other Events which happen in a linear order.
mutwo.core_events.SimultaneousEvent	Event-Object which contains other Event-Objects which happen at the same time.
mutwo.core_events.TaggedSimpleEvent	*SimpleEvent* with tag.
mutwo.core_events.TaggedSequentialEvent	*SequentialEvent* with tag.
mutwo.core_events.TaggedSimultaneousEvent	*SimultaneousEvent* with tag.
mutwo.core_events.Envelope	Model continuous changing values (e.g. glissandi, crescendo).
mutwo.core_events.RelativeEnvelope	Envelope with relative durations and values / parameters.
mutwo.core_events.TempoEnvelope	

`class SimpleEvent`(*duration*, *tempo_envelope=None*)

Bases: *Event*

Event-Object which doesn't contain other Event-Objects (the node or leaf).

Parameters

- **duration** (`core_parameters.abc.Duration`) – The duration of the `SimpleEvent`. Mutwo will convert the incoming object to a *mutwo.core_parameters.abc.Duration* object with the global *core_events.configurations.UNKNOWN_OBJECT_TO_DURATION* callable.
- **tempo_envelope** (*Optional[*`core_events.TempoEnvelope`*]*) –

Example:

```
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(2)
>>> print(simple_event)
SimpleEvent(duration = DirectDuration(2))
```

cut_off(*start*, *end*)

Time-based deletion / shortening of the respective event.

Parameters

- **start** (Duration) – Duration when the cut off shall start.
- **end** (Duration) – Duration when the cut off shall end.

Return type

SimpleEvent

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_off(1, 3)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1)])
```

cut_out(*start*, *end*)

Time-based slicing of the respective event.

Parameters

- **start** (Duration) – Duration when the cut out shall start.
- **end** (Duration) – Duration when the cut up shall end.

Return type

SimpleEvent

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_out(1, 4)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 2), SimpleEvent(duration = 1)])
```

destructive_copy()

Adapted deep copy method that returns a new object for every leaf.

It's called 'destructive', because it forgets potential repetitions of the same object in compound objects. Instead of reproducing the original structure of the compound object that shall be copied, every repetition of the same reference will return a new unique independent object.

The following example shall illustrate the difference between copy.deepcopy and destructive_copy:

```
>>> import copy
>>> from mutwo import core_events
>>> my_simple_event_0 = core_events.SimpleEvent(2)
>>> my_simple_event_1 = core_events.SimpleEvent(3)
>>> my_sequential_event = core_events.SequentialEvent(
>>>     [my_simple_event_0, my_simple_event_1, my_simple_event_0]
>>> )
>>> deepcopied_event = copy.deepcopy(my_sequential_event)
>>> destructivecopied_event = my_sequential_event.destructive_copy()
>>> deepcopied_event[0].duration = 10  # setting the duration of the first event
>>> destructivecopied_event[0].duration = 10
>>> # return True because the first and the third objects share the same
>>> # reference (both are the same copy of 'my_simple_event_0')
>>> deepcopied_event[0].duration == deepcopied_event[2].duration
True
>>> # return False because destructive_copy forgets the shared reference
>>> destructivecopied_event[0].duration == destructivecopied_event[2].duration
False
```

Return type
SimpleEvent

`get_parameter`(*parameter_name*, *flat=False*, *filter_undefined=False*)

Return event attribute with the entered name.

Parameters

- **parameter_name** (*str*) – The name of the attribute that shall be returned.
- **flat** (*filter_undefined*) – `True` for flat sequence of parameter values, `False` if the resulting `tuple` shall repeat the nested structure of the event.
- **filter_undefined** (*bool*) – If set to `True` all `None` values will be filtered from the returned tuple. Default to `False`. This flag has no effect on `get_parameter()` of `mutwo.core_events.SimpleEvent`.

Returns
Return tuple containing the assigned values for each contained event. If an event doesn't posses the asked parameter, mutwo will simply add None to the tuple for the respective event.

Return type
Any

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(2), core_events.SimpleEvent(3)]
>>> )
>>> sequential_event.get_parameter('duration')
(2, 3)
>>> simple_event = core_events.SimpleEvent(10)
>>> simple_event.get_parameter('duration')
DirectDuration(10)
>>> simple_event.get_parameter('undefined_parameter')
None
```

`metrize`(*mutate=True*)

Apply tempo envelope of event on itself

Metrize is only syntactic sugar for a call of `EventToMetrizedEvent`:

```
>>> from mutwo import core_converters
>>> core_converters.EventToMetrizedEvent().convert(
>>>     my_event
>>> ) == my_event.metrize()
True
```

Parameters
mutate (*bool*) –

Return type
SimpleEvent

`mutate_parameter`(*parameter_name*, *function*)

Mutate parameter with a function.

Parameters

- **parameter_name** (*str*) – The name of the parameter which shall be mutated.
- **function** (*Union[Callable[[Any], None], Any]*) – The function which mutates the parameter. The function gets as an input the assigned value for the passed parameter_name of the respective object. The function shouldn't return anything, but simply calls a method of the parameter value.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
SimpleEvent

This method is useful when a particular parameter has been assigned to objects that know methods which mutate themselves. Then 'mutate_parameter' is a convenient wrapper to call the methods of those parameters for all children events.

Example:

```
>>> from mutwo import core_events
>>> from mutwo import music_events
>>> from mutwo import music_parameters
>>> sequential_event = core_events.SequentialEvent(
>>>     [
>>>         music_events.NoteLike(
>>>             [
>>>                 music_parameters.WesternPitch('c', 4),
>>>                 music_parameters.WesternPitch('e', 4)],
>>>             ],
>>>             2, 1,
>>>         )
>>>     ]
>>> )
>>> sequential_event.mutate_parameter(
>>>     'pitch_list', lambda pitch_list: [pitch.add(12) for pitch in pitch_list]
>>> )
>>> # now all pitches should be one octave higher (from 4 to 5)
>>> sequential_event.get_parameter('pitch_list')
([WesternPitch(c5), WesternPitch(e5)],)
```

`set_parameter`(*parameter_name*, *object_or_function*, *set_unassigned_parameter=True*)

Sets event parameter to new value.

Parameters

- **parameter_name** (*str*) – The name of the parameter which values shall be changed.
- **object_or_function** (*Union[Callable[[Any], Any], Any]*) – For setting the parameter either a new value can be passed directly or a function can be passed. The function gets as an argument the previous value that has had been assigned to the respective object and has to return a new value that will be assigned to the object.
- **set_unassigned_parameter** (*bool*) – If set to `False` a new parameter will only be assigned to an Event if the Event already has a attribute with the respective *parameter_name*. If the Event doesn't know the attribute yet and *set_unassigned_parameter* is False, the method call will simply be ignored.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type

SimpleEvent

Example:

```
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(2)
>>> simple_event.set_parameter(
>>>     'duration', lambda old_duration: old_duration * 2
>>> )
>>> simple_event.duration
4
>>> simple_event.set_parameter('duration', 3)
>>> simple_event.duration
3
>>> simple_event.set_parameter(
>>>     'unknown_parameter', 10, set_unassigned_parameter=False
>>> )  # this will be ignored
>>> simple_event.unknown_parameter
AttributeError: 'SimpleEvent' object has no attribute 'unknown_parameter'
>>> simple_event.set_parameter(
>>>     'unknown_parameter', 10, set_unassigned_parameter=True
>>> )  # this will be written
```

(continues on next page)

(continued from previous page)

```
>>> simple_event.unknown_parameter
10
```

property duration: *Duration*

The duration of an event.

This has to be an instance of *mutwo.core_parameters.abc.Duration*.

parameter_to_exclude_from_representation_tuple = ('tempo_envelope',)

class SequentialEvent(*iterable=[]*, *tempo_envelope=None*)

Bases: *ComplexEvent*, Generic[T]

Event-Object which contains other Events which happen in a linear order.

Parameters

- **iterable** (*Iterable[T]*) –
- **tempo_envelope** (*Optional[core_events.TempoEnvelope]*) –

cut_off(*start*, *end*)

Time-based deletion / shortening of the respective event.

Parameters

- **start** (*Union[float, Fraction, int]*) – Duration when the cut off shall start.
- **end** (*Union[float, Fraction, int]*) – Duration when the cut off shall end.

Return type

SequentialEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_off(1, 3)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1)])
```

cut_out(*start*, *end*)

Time-based slicing of the respective event.

Parameters

- **start** (*Union[float, Fraction, int]*) – Duration when the cut out shall start.
- **end** (*Union[float, Fraction, int]*) – Duration when the cut up shall end.

Return type

SequentialEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_out(1, 4)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 2), SimpleEvent(duration = 1)])
```

get_event_at(*absolute_time*)

Get event which is active at the passed absolute_time.

Parameters

absolute_time (*Union[core_parameters.abc.Duration, Any]*) – The absolute time where the method shall search for the active event.

Returns
Event if there is any event at the requested absolute time and `None` if there isn't any event.

Return type
Optional[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(2), core_events.
→SimpleEvent(3)])
>>> sequential_event.get_event_at(1)
SimpleEvent(duration = 2)
>>> sequential_event.get_event_at(3)
SimpleEvent(duration = 3)
>>> sequential_event.get_event_at(100)
None
```

get_event_index_at(*absolute_time*)

Get index of event which is active at the passed absolute_time.

Parameters
absolute_time (*Union[core_parameters.abc.Duration, Any]*) – The absolute time where the method shall search for the active event.

Returns
Index of event if there is any event at the requested absolute time and `None` if there isn't any event.

Return type
Optional[int]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(2), core_events.
→SimpleEvent(3)])
>>> sequential_event.get_event_index_at(1)
0
>>> sequential_event.get_event_index_at(3)
1
>>> sequential_event.get_event_index_at(100)
None
```

split_child_at(*absolute_time*)

Split child event in two events at `absolute_time`.

Parameters
- **absolute_time** (*Union[Duration, Any]*) – where child event shall be split
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
SequentialEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.split_child_at(1)
>>> sequential_event
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 2)])
```

squash_in(*start*, *event_to_squash_in*)

Time-based insert of a new event into the present event.

Parameters
- **start** (*Union[Duration, Any]*) – Absolute time where the event shall be inserted.

- **event_to_squash_in** (Event) – the event that shall be squashed into the present event.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
SequentialEvent[*T*]

Squash in a new event to the present event.

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.squash_in(1, core_events.SimpleEvent(1.5))
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1.5), SimpleEvent(duration = 0.5)])
```

property absolute_time_tuple: tuple[Union[float, fractions.Fraction, int], ...]

Return absolute point in time for each event.

property duration: *Duration*

The duration of an event.

This has to be an instance of *mutwo.core_parameters.abc.Duration*.

property start_and_end_time_per_event: tuple[ranges.ranges.Range, ...]

Return start and end time for each event.

class SimultaneousEvent(*iterable=[]*, *tempo_envelope=None*)

Bases: *ComplexEvent*, `Generic[T]`

Event-Object which contains other Event-Objects which happen at the same time.

Parameters

- **iterable** (*Iterable[T]*) –
- **tempo_envelope** (*Optional[core_events.TempoEnvelope]*) –

cut_off(*start*, *end*)

Time-based deletion / shortening of the respective event.

Parameters

- **start** (*Union[float, Fraction, int]*) – Duration when the cut off shall start.
- **end** (*Union[float, Fraction, int]*) – Duration when the cut off shall end.

Return type
SimultaneousEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_off(1, 3)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1)])
```

cut_out(*start*, *end*)

Time-based slicing of the respective event.

Parameters

- **start** (*Union[Duration, Any]*) – Duration when the cut out shall start.
- **end** (*Union[Duration, Any]*) – Duration when the cut up shall end.

Return type
SimultaneousEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_out(1, 4)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 2), SimpleEvent(duration = 1)])
```

split_child_at(*absolute_time*)

Split child event in two events at `absolute_time`.

Parameters

- **absolute_time** (*Union[float, Fraction, int]*) – where child event shall be split
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type

SimultaneousEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.split_child_at(1)
>>> sequential_event
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 2)])
```

squash_in(*start*, *event_to_squash_in*)

Time-based insert of a new event into the present event.

Parameters

- **start** (*Union[Duration, Any]*) – Absolute time where the event shall be inserted.
- **event_to_squash_in** (*Event*) – the event that shall be squashed into the present event.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type

SimultaneousEvent[*T*]

Squash in a new event to the present event.

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.squash_in(1, core_events.SimpleEvent(1.5))
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1.5), SimpleEvent(duration = 0.5)])
```

property duration: Union[float, Fraction, int]

The duration of an event.

This has to be an instance of *mutwo.core_parameters.abc.Duration*.

class TaggedSimpleEvent(**args*, *tag=None*, ***kwargs*)

Bases: *SimpleEvent*

SimpleEvent with tag.

Parameters

tag (*Optional[str]*) –

class TaggedSequentialEvent(**args*, *tag=None*, ***kwargs*)

Bases: *SequentialEvent*, `Generic[T]`

SequentialEvent with tag.

Parameters
: **tag** (*Optional[str]*) –

class TaggedSimultaneousEvent(**args, tag=None, **kwargs*)

Bases: *SimultaneousEvent*, Generic[T]

SimultaneousEvent with tag.

Parameters
: **tag** (*Optional[str]*) –

class Envelope(*event_iterable_or_point_sequence, tempo_envelope=None, event_to_parameter=<function Envelope.<lambda>>, event_to_curve_shape=<function Envelope.<lambda>>, parameter_to_value=<function Envelope.<lambda>>, value_to_parameter=<function Envelope.<lambda>>, apply_parameter_on_event=<function Envelope.<lambda>>, apply_curve_shape_on_event=<function Envelope.<lambda>>, default_event_class=<class 'mutwo.core_events.basic.SimpleEvent'>, initialise_default_event_class=<function Envelope.<lambda>>*)

Bases: *SequentialEvent*, Generic[T]

Model continuous changing values (e.g. glissandi, crescendo).

Parameters

- **event_iterable_or_point_sequence** (*Iterable[T]*) – An iterable filled with events or with points. If the sequence is filled with points, the points will be converted to events. Each event represents a point in a two dimensional graph where the x-axis presents time and the y-axis a changing value. Any event class can be used. It is more important that the used event classes fit with the functions passed in the following parameters.
- **event_to_parameter** (*Callable[[core_events.abc.Event], core_constants.ParameterType]*) – A function which receives an event and has to return a parameter object (any object). By default the function will ask the event for its *value* property. If the property can't be found it will return 0.
- **event_to_curve_shape** (*Callable[[core_events.abc.Event], CurveShape]*) – A function which receives an event and has to return a curve_shape. A curve_shape is either a float, an integer or a fraction. For a curve_shape = 0 a linear transition between two points is created. For a curve_shape > 0 the envelope changes slower at the beginning and faster at the end, for a curve_shape < 0 it is the inverse behaviour. The default function will ask the event for its *curve_shape* property. If the property can't be found it will return 0.
- **parameter_to_value** (*Callable[[Value], core_constants.ParameterType]*) – Convert a parameter to a value. A value is any object which supports mathematical operations.
- **value_to_parameter** (*Callable[[Value], core_constants.ParameterType]*) – A callable object which converts a value to a parameter.
- **apply_parameter_on_event** (*Callable[[core_events.abc.Event, core_constants.ParameterType], None]*) – A callable object which applies a parameter on an event.
- **apply_curve_shape_on_event** (*Callable[[core_events.abc.Event, CurveShape], None]*) – A callable object which applies a curve shape on an event.
- **default_event_class** (*type[core_events.abc.Event]*) – The default event class which describes a point.
- **initialise_default_event_class** (*Callable[[type[core_events.abc.Event], core_constants.DurationType], core_events.abc.Event]*) –
- **tempo_envelope** (*Optional[core_events.TempoEnvelope]*) –

This class is inspired by Marc Evansteins *Envelope* class in his expenvelope python package and is made to fit better into the *mutwo* ecosystem.

Example:

```
>>> from mutwo import core_events
>>> core_events.Envelope([[0, 0, 1], [0.5, 1]])
Envelope([SimpleEvent(curve_shape = 1, duration = 0.5, value = 0), SimpleEvent(curve_shape = 0, duration =␣
→0.0, value = 1)])
```

CompletePoint
: alias of tuple[Union[float, Fraction, int], Any, Union[float, Fraction, int]]

IncompletePoint
: alias of tuple[Union[float, Fraction, int], Any]

classmethod from_points(**point*, ***kwargs*)

Parameters
`point` (*Point*) –

Return type
Envelope

get_average_parameter(*start=None*, *end=None*)

Parameters

- `start` (*Optional[Union[float, Fraction, int]]*) –
- `end` (*Optional[Union[float, Fraction, int]]*) –

Return type
Any

get_average_value(*start=None*, *end=None*)

Parameters

- `start` (*Optional[Union[*`core_parameters.abc.Duration`*, Any]]*) –
- `end` (*Optional[Union[*`core_parameters.abc.Duration`*, Any]]*) –

Return type
Value

integrate_interval(*start*, *end*)

Parameters

- `start` (*Union[float, Fraction, int]*) –
- `end` (*Union[float, Fraction, int]*) –

Return type
float

parameter_at(*absolute_time*)

Parameters
`absolute_time` (*Union[float, Fraction, int]*) –

Return type
Any

value_at(*absolute_time*)

Parameters
`absolute_time` (*Union[*`core_parameters.abc.Duration`*, Any]*) –

Return type
Value

CurveShape
alias of `Union[float, Fraction, int]`

Point
alias of `Union[tuple[Union[float, Fraction, int], Any, Union[float, Fraction, int]], tuple[Union[float, Fraction, int], Any]]`

Value
alias of `Union[float, Fraction, int]`

property curve_shape_tuple: tuple[CurveShape, ...]

property is_static: bool
Return *True* if `Envelope` only has one static value.

property parameter_tuple: tuple[Any, ...]

property value_tuple: tuple[Value, ...]

class RelativeEnvelope(**args, base_parameter_and_relative_parameter_to_absolute_parameter, **kwargs*)

Bases: *Envelope*, Generic[T]

Envelope with relative durations and values / parameters.

Parameters

- **event_iterable_or_point_sequence** (*Iterable[T]*) – An iterable filled with events or with points. If the sequence is filled with points, the points will be converted to events. Each event represents a point in a two dimensional graph where the x-axis presents time and the y-axis a changing value. Any event class can be used. It is more important that the used event classes fit with the functions passed in the following parameters.
- **event_to_parameter** (*Callable[[core_events.abc.Event], core_constants.ParameterType]*) – A function which receives an event and has to return a parameter object (any object). By default the function will ask the event for its *value* property. If the property can't be found it will return 0.
- **event_to_curve_shape** (*Callable[[core_events.abc.Event], CurveShape]*) – A function which receives an event and has to return a curve_shape. A curve_shape is either a float, an integer or a fraction. For a curve_shape = 0 a linear transition between two points is created. For a curve_shape > 0 the envelope changes slower at the beginning and faster at the end, for a curve_shape < 0 it is the inverse behaviour. The default function will ask the event for its *curve_shape* property. If the property can't be found it will return 0.
- **parameter_to_value** (*Callable[[Value], core_constants.ParameterType]*) – Convert a parameter to a value. A value is any object which supports mathematical operations.
- **value_to_parameter** (*Callable[[Value], core_constants.ParameterType]*) – A callable object which converts a value to a parameter.
- **apply_parameter_on_event** (*Callable[[core_events.abc.Event, core_constants.ParameterType], None]*) – A callable object which applies a parameter on an event.
- **apply_curve_shape_on_event** (*Callable[[core_events.abc.Event, CurveShape], None]*) – A callable object which applies a curve shape on an event.
- **default_event_class** (*type[core_events.abc.Event]*) – The default event class which describes a point.
- **initialise_default_event_class** (*Callable[[type[core_events.abc.Event], core_constants.DurationType], core_events.abc.Event]*) –
- **base_parameter_and_relative_parameter_to_absolute_parameter** (*Callable[[core_constants.ParameterType, core_constants.ParameterType], core_constants.ParameterType]*) – A function which runs when the *resolve()* is called. It expects the base parameter and the relative parameter (which is extracted from the envelope events) and should return an absolute parameter.

This class is inspired by Marc Evansteins *Envelope* class in his expenvelope python package and is made to fit better into the *mutwo* ecosystem.

Example:

```
>>> from mutwo import core_events
>>> core_events.Envelope([[0, 0, 1], [0.5, 1]])
Envelope([SimpleEvent(curve_shape = 1, duration = 0.5, value = 0), SimpleEvent(curve_shape = 0, duration =␣
→0.0, value = 1)])
```

The *RelativeEnvelope* adds the *resolve()* method to the base class *Envelope*.

resolve(*duration, base_parameter, resolve_envelope_class=<class 'mutwo.core_events.envelopes.Envelope'>*)

Parameters

- **duration** (*Union[Duration, Any]*) –
- **base_parameter** (*Any*) –
- **resolve_envelope_class** (*type[mutwo.core_events.envelopes.Envelope]*) –

Return type

Envelope

class TempoEnvelope(*event_iterable_or_point_sequence, tempo_envelope=None, event_to_parameter=<function Envelope.<lambda>>, event_to_curve_shape=<function Envelope.<lambda>>, parameter_to_value=<function Envelope.<lambda>>, value_to_parameter=<function Envelope.<lambda>>, apply_parameter_on_event=<function Envelope.<lambda>>, apply_curve_shape_on_event=<function Envelope.<lambda>>, default_event_class=<class 'mutwo.core_events.basic.SimpleEvent'>, initialise_default_event_class=<function Envelope.<lambda>>*)

Bases: *Envelope*

Parameters

- **event_iterable_or_point_sequence** (*Union[Iterable[T], Sequence[Point]]*) –
- **tempo_envelope** (*Optional[core_events.TempoEnvelope]*) –
- **event_to_parameter** (*Callable[[core_events.abc.Event], core_constants.ParameterType]*) –
- **event_to_curve_shape** (*Callable[[core_events.abc.Event], CurveShape]*) –
- **parameter_to_value** (*Callable[[Value], core_constants.ParameterType]*) –
- **value_to_parameter** (*Callable[[Value], core_constants.ParameterType]*) –
- **apply_parameter_on_event** (*Callable[[core_events.abc.Event, core_constants.ParameterType], None]*) –
- **apply_curve_shape_on_event** (*Callable[[core_events.abc.Event, CurveShape], None]*) –
- **default_event_class** (*type[core_events.abc.Event]*) –
- **initialise_default_event_class** (*Callable[[type[core_events.abc.Event], core_constants.DurationType], core_events.abc.Event]*) –

mutwo.core_events.abc

Abstract base classes for events (definition of public API).

`class ComplexEvent`(*iterable=[]*, *tempo_envelope=None*)

Bases: *Event*, ABC, list[T], Generic[T]

Abstract Event-Object, which contains other Event-Objects.

Parameters

- **iterable** (*Iterable[T]*) –
- **tempo_envelope** (*Optional[core_events.TempoEnvelope]*) –

`destructive_copy()`

Adapted deep copy method that returns a new object for every leaf.

It's called 'destructive', because it forgets potential repetitions of the same object in compound objects. Instead of reproducing the original structure of the compound object that shall be copied, every repetition of the same reference will return a new unique independent object.

The following example shall illustrate the difference between copy.deepcopy and destructive_copy:

```
>>> import copy
>>> from mutwo import core_events
>>> my_simple_event_0 = core_events.SimpleEvent(2)
>>> my_simple_event_1 = core_events.SimpleEvent(3)
>>> my_sequential_event = core_events.SequentialEvent(
>>>     [my_simple_event_0, my_simple_event_1, my_simple_event_0]
>>> )
>>> deepcopied_event = copy.deepcopy(my_sequential_event)
>>> destructivecopied_event = my_sequential_event.destructive_copy()
>>> deepcopied_event[0].duration = 10  # setting the duration of the first event
>>> destructivecopied_event[0].duration = 10
>>> # return True because the first and the third objects share the same
>>> # reference (both are the same copy of 'my_simple_event_0')
>>> deepcopied_event[0].duration == deepcopied_event[2].duration
True
>>> # return False because destructive_copy forgets the shared reference
>>> destructivecopied_event[0].duration == destructivecopied_event[2].duration
False
```

Return type

ComplexEvent[*T*]

empty_copy()

Make a copy of the *ComplexEvent* without any child events.

This method is useful if one wants to copy an instance of *ComplexEvent* and make sure that all side attributes (e.g. any assigned properties specific to the respective subclass) get saved.

Example:

```
>>> from mutwo import core_events
>>> piano_voice_0 = core_events.TaggedSequentialEvent([core_events.SimpleEvent(2)], tag="piano")
>>> piano_voice_1 = piano_voice_0.empty_copy()
>>> piano_voice_1.tag
'piano'
>>> piano_voice_1
TaggedSequentialEvent([])
```

Return type
ComplexEvent[*T*]

filter(*condition*)

Condition-based deletion of child events.

Parameters

- **condition** (*Callable[[Event], bool]*) – Function which takes a *Event* and returns `True` or `False`. If the return value of the function is `False` the respective *Event* will be deleted.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
ComplexEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> simultaneous_event = core_events.SimultaneousEvent(
    [core_events.SimpleEvent(1), core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
)
>>> simultaneous_event.filter(lambda event: event.duration > 2)
>>> simultaneous_event
SimultaneousEvent([SimpleEvent(duration = 3)])
```

get_event_from_index_sequence(*index_sequence*)

Get nested *Event* from a sequence of indices.

Parameters
index_sequence (*Sequence[int]*) – The indices of the nested *Event*.

Return type
Event

Example:

```
>>> from mutwo import core_events
>>> nested_sequential_event = core_events.SequentialEvent(
>>>     [core_events.SequentialEvent([core_events.SimpleEvent(2)])]
>>> )
>>> nested_sequential_event.get_event_from_index_sequence((0, 0))
SimpleEvent(duration = 2)
>>> # this is equal to:
>>> nested_sequential_event[0][0]
SimpleEvent(duration = 2)
```

get_parameter(*parameter_name*, *flat=False*, *filter_undefined=False*)

Return event attribute with the entered name.

Parameters

- **parameter_name** (*str*) – The name of the attribute that shall be returned.
- **flat** (*filter_undefined*) – True for flat sequence of parameter values, False if the resulting tuple shall repeat the nested structure of the event.
- **filter_undefined** (*bool*) – If set to True all None values will be filtered from the returned tuple. Default to False. This flag has no effect on *get_parameter()* of *mutwo.core_events.SimpleEvent*.

Returns

Return tuple containing the assigned values for each contained event. If an event doesn't posses the asked parameter, mutwo will simply add None to the tuple for the respective event.

Return type

tuple[*Any*, ...]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(2), core_events.SimpleEvent(3)]
>>> )
>>> sequential_event.get_parameter('duration')
(2, 3)
>>> simple_event = core_events.SimpleEvent(10)
>>> simple_event.get_parameter('duration')
DirectDuration(10)
>>> simple_event.get_parameter('undefined_parameter')
None
```

metrize(*mutate=True*)

Apply tempo envelope of event on itself

Metrize is only syntactic sugar for a call of EventToMetrizedEvent:

```
>>> from mutwo import core_converters
>>> core_converters.EventToMetrizedEvent().convert(
>>>     my_event
>>> ) == my_event.metrize()
True
```

Parameters

mutate (*bool*) –

Return type

ComplexEvent

mutate_parameter(*parameter_name*, *function*)

Mutate parameter with a function.

Parameters

- **parameter_name** (*str*) – The name of the parameter which shall be mutated.
- **function** (*Union[Callable[[Any], None], Any]*) – The function which mutates the parameter. The function gets as an input the assigned value for the passed parameter_name of the respective object. The function shouldn't return anything, but simply calls a method of the parameter value.
- **mutate** – If False the function will return a copy of the given object. If set to True the object itself will be changed and the function will return the changed object. Default to True.

Return type

ComplexEvent[*T*]

This method is useful when a particular parameter has been assigned to objects that know methods which mutate themselves. Then 'mutate_parameter' is a convenient wrapper to call the methods of those parameters for all children events.

Example:

```
>>> from mutwo import core_events
>>> from mutwo import music_events
>>> from mutwo import music_parameters
>>> sequential_event = core_events.SequentialEvent(
>>>     [
>>>         music_events.NoteLike(
>>>             [
>>>                 music_parameters.WesternPitch('c', 4),
>>>                 music_parameters.WesternPitch('e', 4)],
>>>             ],
>>>             2, 1,
>>>         )
>>>     ]
>>> )
>>> sequential_event.mutate_parameter(
>>>     'pitch_list', lambda pitch_list: [pitch.add(12) for pitch in pitch_list]
>>> )
>>> # now all pitches should be one octave higher (from 4 to 5)
>>> sequential_event.get_parameter('pitch_list')
([WesternPitch(c5), WesternPitch(e5)],)
```

set_parameter(*parameter_name*, *object_or_function*, *set_unassigned_parameter=True*)

Sets parameter to new value for all children events.

Parameters

- **parameter_name** (*str*) – The name of the parameter which values shall be changed.
- **object_or_function** (*Union[Callable[[Any], Any], Any]*) – For setting the parameter either a new value can be passed directly or a function can be passed. The function gets as an argument the previous value that has had been assigned to the respective object and has to return a new value that will be assigned to the object.
- **set_unassigned_parameter** (*bool*) – If set to False a new parameter will only be assigned to an Event if the Event already has a attribute with the respective *parameter_name*. If the Event doesn't know the attribute yet and *set_unassigned_parameter* is False, the method call will simply be ignored.
- **mutate** – If False the function will return a copy of the given object. If set to True the object itself will be changed and the function will return the changed object. Default to True.

Returns

The event.

Return type

ComplexEvent[*T*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(2), core_events.SimpleEvent(3)]
>>> )
>>> sequential_event.set_parameter('duration', lambda duration: duration * 2)
>>> sequential_event.get_parameter('duration')
(4, 6)
```

abstract split_child_at(*absolute_time*)

Split child event in two events at absolute_time.

Parameters

- **absolute_time** (Duration) – where child event shall be split
- **mutate** – If False the function will return a copy of the given object. If set to True the object itself will be changed and the function will return the changed object. Default to True.

Return type

Optional[ComplexEvent[*T*]]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.split_child_at(1)
>>> sequential_event
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 2)])
```

abstract squash_in(*start*, *event_to_squash_in*)

Time-based insert of a new event into the present event.

Parameters

- **start** (Duration) – Absolute time where the event shall be inserted.
- **event_to_squash_in** (Event) – the event that shall be squashed into the present event.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type

Optional[ComplexEvent[*T*]]

Squash in a new event to the present event.

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.squash_in(1, core_events.SimpleEvent(1.5))
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1.5), SimpleEvent(duration = 0.5)])
```

tie_by(*condition*, *process_surviving_event=<function ComplexEvent.<lambda>>*, *event_type_to_examine=<class 'mutwo.core_events.abc.Event'>*, *event_to_remove=True*)

Condition-based deletion of neighboring child events.

Parameters

- **condition** (*Callable[[*Event*,* Event*],* bool*]*) – Function which compares two neighboring events and decides whether one of those events shall be removed. The function should return *True* for deletion and *False* for keeping both events.
- **process_surviving_event** (*Callable[[*Event*,* Event*],* None*]*) – Function which gets two arguments: first the surviving event and second the event which shall be removed. The function should process the surviving event depending on the removed event. By default, mutwo will simply add the *duration* of the removed event to the duration of the surviving event.
- **event_type_to_examine** (*Type[*Event*]*) – Defines which events shall be compared. If one only wants to process the leaves, this should perhaps be *mutwo.core_events.SimpleEvent*.
- **event_to_remove** (*bool*) – *True* if the second (left) event shall be removed and *False* if the first (right) event shall be removed.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type

ComplexEvent[*T*]

abstract property duration: *Duration*

The duration of an event.

This has to be an instance of *mutwo.core_parameters.abc.Duration*.

class Event(*tempo_envelope=None*)

Bases: `ABC`

Abstract Event-Object

Parameters

tempo_envelope (*Optional[*core_events.TempoEnvelope*]*) – An envelope which describes the dynamic tempo of an event.

copy()

Return a deep copy of the given Event.

Return type

Event

abstract cut_off(*start*, *end*)

Time-based deletion / shortening of the respective event.

Parameters

- **start** (Duration) – Duration when the cut off shall start.
- **end** (Duration) – Duration when the cut off shall end.

Return type

Optional[Event]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_off(1, 3)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 1), SimpleEvent(duration = 1)])
```

abstract cut_out(*start*, *end*)

Time-based slicing of the respective event.

Parameters

- **start** (Duration) – Duration when the cut out shall start.
- **end** (Duration) – Duration when the cut up shall end.

Return type

Optional[Event]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(3), core_events.SimpleEvent(2)]
>>> )
>>> sequential_event.cut_out(1, 4)
>>> print(sequential_event)
SequentialEvent([SimpleEvent(duration = 2), SimpleEvent(duration = 1)])
```

abstract destructive_copy()

Adapted deep copy method that returns a new object for every leaf.

It's called 'destructive', because it forgets potential repetitions of the same object in compound objects. Instead of reproducing the original structure of the compound object that shall be copied, every repetition of the same reference will return a new unique independent object.

The following example shall illustrate the difference between copy.deepcopy and destructive_copy:

```
>>> import copy
>>> from mutwo import core_events
>>> my_simple_event_0 = core_events.SimpleEvent(2)
>>> my_simple_event_1 = core_events.SimpleEvent(3)
>>> my_sequential_event = core_events.SequentialEvent(
>>>     [my_simple_event_0, my_simple_event_1, my_simple_event_0]
>>> )
>>> deepcopied_event = copy.deepcopy(my_sequential_event)
>>> destructivecopied_event = my_sequential_event.destructive_copy()
>>> deepcopied_event[0].duration = 10  # setting the duration of the first event
>>> destructivecopied_event[0].duration = 10
>>> # return True because the first and the third objects share the same
>>> # reference (both are the same copy of 'my_simple_event_0')
>>> deepcopied_event[0].duration == deepcopied_event[2].duration
True
>>> # return False because destructive_copy forgets the shared reference
>>> destructivecopied_event[0].duration == destructivecopied_event[2].duration
False
```

Return type
Event

`abstract get_parameter`(*parameter_name*, *flat=False*, *filter_undefined=False*)

Return event attribute with the entered name.

Parameters

- **parameter_name** (*str*) – The name of the attribute that shall be returned.
- **flat** (*filter_undefined*) – `True` for flat sequence of parameter values, `False` if the resulting `tuple` shall repeat the nested structure of the event.
- **filter_undefined** (*bool*) – If set to `True` all `None` values will be filtered from the returned tuple. Default to `False`. This flag has no effect on *get_parameter()* of *mutwo.core_events.SimpleEvent*.

Returns
Return tuple containing the assigned values for each contained event. If an event doesn't posses the asked parameter, mutwo will simply add None to the tuple for the respective event.

Return type
Union[tuple[*Any*, ...], *Any*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(2), core_events.SimpleEvent(3)]
>>> )
>>> sequential_event.get_parameter('duration')
(2, 3)
>>> simple_event = core_events.SimpleEvent(10)
>>> simple_event.get_parameter('duration')
DirectDuration(10)
>>> simple_event.get_parameter('undefined_parameter')
None
```

`abstract metrize()`

Apply tempo envelope of event on itself

Metrize is only syntactic sugar for a call of `EventToMetrizedEvent`:

```
>>> from mutwo import core_converters
>>> core_converters.EventToMetrizedEvent().convert(
>>>     my_event
>>> ) == my_event.metrize()
True
```

Return type
Optional[Event]

`abstract mutate_parameter`(*parameter_name*, *function*)

Mutate parameter with a function.

Parameters

- **parameter_name** (*str*) – The name of the parameter which shall be mutated.
- **function** (*Union[Callable[[Any], None], Any]*) – The function which mutates the parameter. The function gets as an input the assigned value for the passed parameter_name of the respective object. The function shouldn't return anything, but simply calls a method of the parameter value.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
Optional[Event]

This method is useful when a particular parameter has been assigned to objects that know methods which mutate themselves. Then 'mutate_parameter' is a convenient wrapper to call the methods of those parameters for all children events.

Example:

```
>>> from mutwo import core_events
>>> from mutwo import music_events
>>> from mutwo import music_parameters
>>> sequential_event = core_events.SequentialEvent(
>>>     [
>>>         music_events.NoteLike(
>>>             [
>>>                 music_parameters.WesternPitch('c', 4),
>>>                 music_parameters.WesternPitch('e', 4)],
>>>             ],
>>>             2, 1,
>>>         )
>>>     ]
>>> )
>>> sequential_event.mutate_parameter(
>>>     'pitch_list', lambda pitch_list: [pitch.add(12) for pitch in pitch_list]
>>> )
>>> # now all pitches should be one octave higher (from 4 to 5)
>>> sequential_event.get_parameter('pitch_list')
([WesternPitch(c5), WesternPitch(e5)],)
```

`reset_tempo_envelope()`

Set events tempo envelope so that one beat equals one second (tempo 60).

Parameters
`mutate` – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Return type
Event

Example:

```
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(duration = 1)
>>> simple_event.tempo_envelope[0].value = 100
>>> print(simple_event.tempo_envelope)
TempoEnvelope([SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 1), value = 100),␣
→SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 0), value = 60)])
>>> simple_event.reset_tempo_envelope()
>>> print(simple_event.tempo_envelope)
TempoEnvelope([SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 1), value = 60),␣
→SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 0), value = 60)])
```

`set`(*attribute_name*, *value*)

Set an attribute of the object to a specific value

Parameters

- `attribute_name` (*str*) – The name of the attribute which value shall be set.
- `value` (*Any*) – The value which shall be assigned to the given `attribute_name`
- `mutate` – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Returns
The event.

Return type
Event

This function is merely a convenience wrapper for...

```
>>> event.attribute_name = value
```

Because the function return the event itself it can be used in function composition.

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(2)])
>>> sequential_event.set('duration', 10).set('my_new_attribute', 'hello-world!')
```

`abstract set_parameter`(*parameter_name*, *object_or_function*, *set_unassigned_parameter=True*)

Sets parameter to new value for all children events.

Parameters

- **parameter_name** (*str*) – The name of the parameter which values shall be changed.
- **object_or_function** (*Union[Callable[[Any], Any], Any]*) – For setting the parameter either a new value can be passed directly or a function can be passed. The function gets as an argument the previous value that has had been assigned to the respective object and has to return a new value that will be assigned to the object.
- **set_unassigned_parameter** (*bool*) – If set to False a new parameter will only be assigned to an Event if the Event already has a attribute with the respective *parameter_name*. If the Event doesn't know the attribute yet and *set_unassigned_parameter* is False, the method call will simply be ignored.
- **mutate** – If `False` the function will return a copy of the given object. If set to `True` the object itself will be changed and the function will return the changed object. Default to `True`.

Returns

The event.

Return type

Optional[Event]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent(
>>>     [core_events.SimpleEvent(2), core_events.SimpleEvent(3)]
>>> )
>>> sequential_event.set_parameter('duration', lambda duration: duration * 2)
>>> sequential_event.get_parameter('duration')
(4, 6)
```

`split_at`(*absolute_time*)

Split event in two events at `absolute_time`.

Parameters

absolute_time (Duration) – where event shall be split

Returns

Two events that result from splitting the present event.

Return type

tuple[*mutwo.core_events.abc.Event*, *mutwo.core_events.abc.Event*]

Example:

```
>>> from mutwo import core_events
>>> sequential_event = core_events.SequentialEvent([core_events.SimpleEvent(3)])
>>> sequential_event.split_at(1)
(SequentialEvent([SimpleEvent(duration = 1)]), SequentialEvent([SimpleEvent(duration = 2)]))
>>> sequential_event[0].split_at(1)
(SimpleEvent(duration = 1), SimpleEvent(duration = 2))
```

`abstract property duration:` *Duration*

The duration of an event.

This has to be an instance of *mutwo.core_parameters.abc.Duration*.

`property tempo_envelope:` *TempoEnvelope*

The dynamic tempo of an event; specified as an envelope.

Tempo envelopes are represented as `core_events.TempoEnvelope` objects. Tempo envelopes are valid for its respective event and all its children events.

mutwo.core_events.configurations

Configurations which are shared for all event classes in *mutwo.core_events*.

`UNKNOWN_OBJECT_TO_DURATION`(*unknown_object*)

Global definition of callable to parse objects to *mutwo.core_parameters.abc.Duration*.

This function is used in almost all objects which inherit from *mutwo.core_events.abc.Event*. It implements syntactic sugar so that users can parse buildin types (or other objects) to mutwo callables which expect *mutwo.core_parameters.abc.Duration* objects.

This global variable is the reason why the following code prints a *mutwo.core_parameters.DirectDuration*:

```
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(duration=10)
>>> simple_event.duration
DirectDuration(10)
```

Without this function...

1. It wouldn't be certain that *duration* returns an instance of *mutwo.core_parameters.abc.Duration*.
2. Or the code would raise a `TypeError` and users would be forced to write:

```
>>> core_events.SimpleEvent(core_parameters.DirectDuration(10))
```

Because the syntactic sugar partially violates the Python Zen "Explicit is better than implicit" this function is publicly defined in the *configurations* module (and not in private class methods), so that users are encouraged to override the variable if desired.

`DEFAULT_CURVE_SHAPE_ATTRIBUTE_NAME = 'curve_shape'`

Default attribute name when fetching the curve shape of an event

`DEFAULT_PARAMETER_ATTRIBUTE_NAME = 'value'`

Default attribute name when fetching the parameter of an event

mutwo.core_generators

Table of content

Classes and functions that generate data with the potential of artistic use.

The module is organised in different submodules where each submodule is named after the first known person who introduced the respective algorithms. Unlike the `mutwo.converters` module the entered data and the resulting data can be very different in type and form.

The term 'generators' simply labels the functionality of the module and shouldn't be confused with the Python term for specific functions with the 'yield' keyword.

Object	Documentation
mutwo.core_generators.DynamicChoice	Weighted random choices with dynamically changing weights.

`class DynamicChoice`(*value_sequence*, *curve_sequence*, *random_seed=100*)

Bases: `object`

Weighted random choices with dynamically changing weights.

Parameters

- `value_sequence` (*Sequence[Any]*) – The items to choose from.
- `curve_sequence` (*Sequence[core_events.Envelope]*) – The dynamically changing weight for each value.
- `random_seed` (*int*) – The seed which shall be set at class initialisation.

Example:

```
>>> from mutwo import core_events
>>> from mutwo import core_generators
>>> dynamic_choice = core_generators.DynamicChoice(
>>>    [0, 1, 2],
>>>    [
>>>        core_events.Envelope([(0, 0), (0.5, 1), (1, 0)]),
>>>        core_events.Envelope([(0, 0.5), (0.5, 0), (1, 0.5)]),
>>>        core_events.Envelope([(0, 0.5), (1, 1)]),
>>>    ],
>>> )
>>> dynamic_choice.gamble_at(0.3)
2
>>> dynamic_choice.gamble_at(0.3)
2
>>> dynamic_choice.gamble_at(0.3)
0
```

gamble_at(*time*)

Return value at requested time.

Parameters
`time` (*numbers.Real*) – At which position on the x-Axis shall be gambled.

Returns
The chosen value.

Return type
Any

items()

Return type
tuple[tuple[*Any*, *mutwo.core_events.envelopes.Envelope*]]

mutwo.core_parameters

Table of content

Abstractions for attributes that can be assigned to Event objects.

Object	Documentation
mutwo.core_parameters.DirectDuration	Simple *Duration* which is directly initialised by its value.
mutwo.core_parameters.TempoPoint	Represent the active tempo at a specific moment in time.

class DirectDuration(*duration*)

Bases: *Duration*

Simple *Duration* which is directly initialised by its value.

Example:

```
>>> from mutwo import core_parameters
>>> # create duration with duration = 10 beats
>>> my_duration = core_parameters.DirectDuration(10)
>>> my_duration.duration
10
```

Parameters
`duration` (*float*) –

`property duration: Fraction`

`class TempoPoint`(*tempo_or_tempo_range_in_beats_per_minute*, *reference=1*, *textual_indication=None*)

Bases: `object`

Represent the active tempo at a specific moment in time.

Parameters

- **`tempo_or_tempo_range_in_beats_per_minute`** (*`Union[float, tuple[float, float]]`*) – Specify a tempo in beats per minute. Tempo can also be a tempo range where the first value indicates a minimal tempo and the second value the maximum tempo. If the user specifies a range *mutwo* will use the minimal tempo in internal calculations.
- **`reference`** (*`Union[float, Fraction, int]`*) – The reference with which the tempo will be multiplied. In terms of Western notation a reference = 1 will be a 1/4 beat, a reference of 2 will be a 1/2 beat, etc. Default to 1.
- **`textual_indication`** (*`Optional[str]`*) – Sometimes it is desired to specify an extra text indication how fast or slow the music should be (for instance "Adagio" in Western music). Default to *None*.

Example:

```
>>> from mutwo import core_events
>>> from mutwo import core_parameters
>>> tempo_envelope = core_events.TempoEnvelope([
>>>     [0, core_parameters.TempoPoint(60, reference=2)]
>>> ])
```

`property absolute_tempo_in_beats_per_minute: float`

Get absolute tempo in beats per minute

The absolute tempo takes the `reference` of the *`TempoPoint`* into account.

`property tempo_in_beats_per_minute: float`

Get tempo in beats per minute

If `tempo_or_tempo_range_in_beats_per_minute` is a range mutwo will return the minimal tempo.

mutwo.core_parameters.abc

Abstract base classes for different parameters.

This module defines the public API of parameters. Most other mutwo classes rely on this API. This means when someone creates a new class inheriting from any of the abstract parameter classes which are defined in this module, she or he can make use of all other mutwo modules with this newly created parameter class.

`class Duration`

Bases: *`SingleNumberParameter`*

Abstract base class for any duration.

If the user wants to define a Duration class, the abstract property *`duration`* has to be overridden.

The attribute *`duration`* is stored in unit *beats*.

`add`(*other*)

Parameters
`other` (*`Union[Duration, float, Fraction, int]`*) –

Return type
Duration

`divide`(*other*)

Parameters
`other` (*`Union[Duration, float, Fraction, int]`*) –

Return type
Duration

`multiply`(*other*)

Parameters
: other (*Union[*Duration*, float, Fraction, int]*) –

Return type
: Duration

`subtract`(*other*)

Parameters
: other (*Union[*Duration*, float, Fraction, int]*) –

Return type
: Duration

`direct_comparison_type_tuple = (<class 'float'>, <class 'int'>, <class 'quicktions.Fraction'>)`

`abstract property duration: Fraction`

`property duration_in_floats: float`

`property value_name`

`class ParameterWithEnvelope`(*envelope*)

Bases: `ABC`

Abstract base class for all parameters with an envelope.

Parameters
: envelope (`core_events.RelativeEnvelope`) –

`resolve_envelope`(*duration*, *resolve_envelope_class=None*)

Parameters

- duration (*Union[float, Fraction, int]*) –
- resolve_envelope_class (*Optional[type[*`mutwo.core_events.envelopes.Envelope`*]]*) –

Return type
: Envelope

`property envelope:` *RelativeEnvelope*

`class SingleNumberParameter`

Bases: *SingleValueParameter*

Abstract base class for all parameters which are defined by one number.

Classes which inherit from this base class have to override the same methods and properties as one have to override when inheriting from *SingleValueParameter*.

Furthermore the property *digit_to_round_to_count* can be overridden. This should return an integer or *None*. If it returns an integer it will first round two numbers before comparing them with the == or < or <= or > or >= operators. The default implementation always returns `None.

Example:

```
>>> from mutwo import core_parameters
>>> class Speed(
        core_parameters.abc.SingleNumberParameter,
        value_name="meter_per_seconds",
        value_return_type=float
    ):
        def __init__(self, meter_per_seconds: float):
            self._meter_per_seconds = meter_per_seconds
        @property
        def meter_per_seconds(self) -> float:
            return self._meter_per_seconds
>>> light_speed = Speed(299792458)
>>> sound_speed = Speed(343)
>>> light_speed > sound_speed
True
```

property digit_to_round_to_count: Optional[int]

direct_comparison_type_tuple = ()

class SingleValueParameter

Bases: ABC

Abstract base class for all parameters which are defined by one value.

Classes which inherit from this base class have to provide an additional keyword argument *value_name*. Furthermore they can provide the optional keyword argument *value_return_type*.

Example:

```
>>> from mutwo import core_parameters
>>> class Color(
        core_parameters.abc.SingleValueParameter,
        value_name="color",
        value_return_type=str
    ):
        def __init__(self, color: str):
            self._color = color
        @property
        def color(self) -> str:
            return self._color
>>> red = Color('red')
>>> red.color
'red'
>>> orange = Color('orange')
>>> red2 = Color('red')
>>> red == orange
False
>>> red == red2
True
```

mutwo.core_parameters.configurations

Configurations which are shared for all parameter classes in *mutwo.core_parameters*.

ROUND_DURATION_TO_N_DIGITS = 10

Set floating point precision for the *duration_in_floats* property of all *Duration* classes in the *mutwo.core_parameters* module.

When returning the *duration_in_floats* property all mentioned events will round their actual duration if the duration type is float. This behaviour has been added with version 0.28.1 to avoid floating point rounding errors which could occur in all duration related methods of the different event classes (as it can happen in for instance the *mutwo.core_events.abc.ComplexEvent.squash_in()* method or the *mutwo.core_events.abc.Event.cut_off()* method).

mutwo.core_utilities

Table of content

Utility functions.

Object	Documentation
mutwo.core_utilities.add_copy_option	This decorator adds a copy option for object mutating methods.
mutwo.core_utilities.add_tag_to_class	This decorator adds a 'tag' argument to the init method of a class.
mutwo.core_utilities.compute_lazy	Cache function output to disk via pickle.
mutwo.core_utilities.AlreadyDefinedValueNameError	

continues on next page

Table 2 – continued from previous page

Object	Documentation
mutwo.core_utilities.InvalidAverageValueStartAndEndWarning	
mutwo.core_utilities.InvalidStartValueError	
mutwo.core_utilities.InvalidPointError	
mutwo.core_utilities.ImpossibleToSquashInError	
mutwo.core_utilities.InvalidStartAndEndValueError	
mutwo.core_utilities.InvalidCutOutStartAndEndValuesError	
mutwo.core_utilities.SplitUnavailableChildError	
mutwo.core_utilities.NoSolutionFoundError	
mutwo.core_utilities.factorise	factorise(integer) -> [list of factors]
mutwo.core_utilities.factors	Get factor generator
mutwo.core_utilities.is_prime	Test if number is prime or not.
mutwo.core_utilities.scale	Scale a value from one range to another range.
mutwo.core_utilities.scale_sequence_to_sum	Scale numbers in a sequence so that the resulting sum fits to the given value.
mutwo.core_utilities.accumulate_from_n	Accumulates iterable starting with value n.
mutwo.core_utilities.accumulate_from_zero	Accumulates iterable starting from 0.
mutwo.core_utilities.insert_next_to	Insert an item into a list relative to the first item equal to a certain value.
mutwo.core_utilities.uniqify_sequence	Not-Order preserving function to uniqify any iterable with non-hashable objects.
mutwo.core_utilities.cyclic_permutations	Cyclic permutation of an iterable. Return a generator object.
mutwo.core_utilities.find_closest_index	Return index of element in `data` with smallest difference to `item`.
mutwo.core_utilities.find_closest_item	Return element in `data` with smallest difference to `item`.
mutwo.core_utilities.get_nested_item_from_index_sequence	Get item in nested Sequence.
mutwo.core_utilities.set_nested_item_from_index_sequence	Set item in nested Sequence.
mutwo.core_utilities.find_numbers_which_sums_up_to	Find all combinations of numbers which sum is equal to the given sum.
mutwo.core_utilities.call_function_except_attribute_error	Run a function with argument as input
mutwo.core_utilities.round_floats	Round number if it is an instance of float, otherwise unaltered number.
mutwo.core_utilities.camel_case_to_snake_case	Transform camel case formatted string to snake case.
mutwo.core_utilities.test_if_objects_are_equal_by_parameter_tuple	Check if the parameters of two objects have equal values.
mutwo.core_utilities.get_all	Fetch from all arguments their __*all*__ attribute and combine them to one tuple

add_copy_option(*function*)

This decorator adds a copy option for object mutating methods.

Parameters

- **function** (*F*) – The method which shall be adjusted.
- **function** –

Return type

F

The 'add_copy_option' decorator adds the 'mutate' keyword argument to the decorated method. If 'mutate' is set to `False`, the decorator deep copies the respective object, then applies the called method on the new copied object and finally returns the copied object. This can be useful for methods that by default mutate its object. When adding this method, it is up to the user whether the original object shall be changed and returned (for mutate=True) or if a copied version of the object with the respective mutation shall be returned (for mutate=False).

add_tag_to_class(*class_to_decorate*)

This decorator adds a 'tag' argument to the init method of a class.

Parameters

- **class_to_decorate** (*G*) – The class which shall be decorated.

- `class_to_decorate` –

Return type
G

`compute_lazy`(*path*, *force_to_compute=False*, *pickle_module=None*)

Cache function output to disk via pickle.

Parameters

- **path** (*str*) – Where to save the computed result.
- **force_to_compute** (*bool*) – Set to `True` if function has to be re-computed.
- **pickle_module** (*Optional[types.ModuleType]*) – Depending on the object which should be pickled the default python pickle module won't be sufficient. Therefore alternative third party pickle modules (with the same API) can be used. If no argument is provided, the function will first try to use any of the pickle modules given in the *mutwo.core_utilities.configurations.PICKLE_MODULE_TO_SEARCH_TUPLE*. If none of the modules could be imported it will fall back to the buildin pickle module.

The decorator will only run the function if its input changes and otherwise load the return value from the disk.

This function is helpful if there is a complex, long-taking calculation, which should only run once or from time to time if the input changes.

Example:

```
>>> from mutwo.utilities import decorators
>>> @decorators.compute_lazy("magic_output", False)
    def my_super_complex_calculation(n_numbers):
        return sum(number for number in range(n_numbers))
>>> N_NUMBERS = 100000000
>>> my_super_complex_calculation(N_NUMBERS)
4999999950000000
>>> # takes very little time when calling the function the second time
>>> my_super_complex_calculation(N_NUMBERS)
4999999950000000
>>> # takes long again, because the input changed
>>> my_super_complex_calculation(N_NUMBERS + 10)
4999999950000000
```

`factorise`(*integer*) → [list of factors]

Parameters
number_to_factorise (*int*) – The number which shall be factorised.

Returns
Returns a list of the (mostly) prime factors of integer n. For negative integers, -1 is included as a factor. If n is 0, 1 or -1, [n] is returned as the only factor. Otherwise all the factors will be prime.

Return type
list[int]

Example:

```
>>> factorise(-693)
[-1, 3, 3, 7, 11]
>>> factorise(55614)
[2, 3, 13, 23, 31]
```

`factors`(*number*)

Get factor generator

Parameters
number (*int*) – The number from which to yield factors.

Return type
Generator

Yields tuples of (factor, count) where each factor is unique and usually prime, and count is an integer 1 or larger. The factors are prime, except under the following circumstances: if the argument n is negative, -1 is included as a factor; if n is 0 or 1, it is given as the only factor. For all other integer n, all of the factors returned are prime.

Example:

```
>>> list(factors(3*7*7*7*11))
[(3, 1), (7, 3), (11, 1)]
```

is_prime(*number_to_test*)

Test if number is prime or not.

Parameters

number_to_test (*int*) – The number which shall be tested.

Returns

True if number is prime and False if number isn't a Prime.

Return type

bool

(has been copied from here)

scale(*value*, *old_min*, *old_max*, *new_min*, *new_max*, *translation_shape=0*)

Scale a value from one range to another range.

Parameters

- **value** (*Union[float, Fraction, int]*) – The value that shall be scaled.
- **old_min** (*Union[float, Fraction, int]*) – The minima of the old range.
- **old_max** (*Union[float, Fraction, int]*) – The maxima of the old range.
- **new_min** (*Union[float, Fraction, int]*) – The minima of the new range.
- **new_max** (*Union[float, Fraction, int]*) – The maxima of the new range.
- **translation_shape** (*Union[float, Fraction, int]*) – 0 for a linear translation, values > 0 for a slower change at the beginning, values < 0 for a faster change at the beginning.

Return type

Union[float, *Fraction*, int]

The algorithmic to change the translation with the *translation_shape* has been copied from expenvelope by M. Evanstein.

Example:

```
>>> from mutwo.core.utilities import tools
>>> tools.scale(1, 0, 1, 0, 100)
100
>>> tools.scale(0.5, 0, 1, 0, 100)
50
>>> tools.scale(0.2, 0, 1, 0, 100)
20
>>> tools.scale(0.2, 0, 1, 0, 100, 1)
12.885124808584155
>>> tools.scale(0.2, 0, 1, 0, 100, -1)
28.67637263023771
```

scale_sequence_to_sum(*sequence_to_scale*, *sum_to_scale_to*)

Scale numbers in a sequence so that the resulting sum fits to the given value.

Parameters

- **sequence_to_scale** (*Sequence[core_constants.Real]*) – The sequence filled with real numbers which sum should fit to the given *sum_to_scale_to* argument.
- **sum_to_scale_to** (*core_constants.Real*) – The resulting sum of the sequence.

Return type

Sequence[*Union*[float, *Fraction*, int]]

Example:

```
>>> from mutwo import utilities
>>> sequence_to_scale = [1, 3, 2]
>>> utilities.tools.scale_sequence_to_sum(sequence_to_scale, 3)
[0.5, 1.5, 1]
```

accumulate_from_n(*iterable*, *n*)

Accumulates iterable starting with value n.

Parameters

- **iterable** (*Iterable[Union[float, Fraction, int]]*) – The iterable which values shall be accumulated.
- **n** (*Union[float, Fraction, int]*) – The start number from which shall be accumulated.

Return type

Iterator

Example:

```
>>> from mutwo.utilities import tools
>>> tools.accumulate_from_n((4, 2, 3), 0)
(0, 4, 6, 9)
>>> tools.accumulate_from_n((4, 2, 3), 2)
(2, 6, 8, 11)
```

accumulate_from_zero(*iterable*)

Accumulates iterable starting from 0.

Parameters

iterable (*Iterable[Union[float, Fraction, int]]*) – The iterable which values shall be accumulated.

Return type

Iterator

Example:

```
>>> from mutwo.utilities import tools
>>> tools.accumulate_from_zero((4, 2, 3), 0)
(0, 4, 6, 9)
```

insert_next_to(*mutable_sequence*, *item_to_find*, *distance*, *item_to_insert*)

Insert an item into a list relative to the first item equal to a certain value.

Parameters

- **mutable_sequence** (*MutableSequence*) –
- **item_to_find** (*Any*) –
- **distance** (*int*) –
- **item_to_insert** (*Any*) –

uniqify_sequence(*sequence*, *sort_key=None*, *group_by_key=None*)

Not-Order preserving function to uniqify any iterable with non-hashable objects.

Parameters

- **sequence** (*Sequence*) – The iterable which items shall be uniqified.
- **sort_key** (*Optional[Callable[[Any], Union[float, Fraction, int]]]*) –
- **group_by_key** (*Optional[Callable[[Any], Any]]*) –

Returns

Return uniqified version of the entered iterable. The function will try to return the same type of the passed iterable. If Python raises an error during initialisation of the original iterable type, the function will simply return a tuple.

Return type

Iterable

Example:

```
>>> from mutwo.parameters import pitches
>>> from mutwo.utilities import tools
>>> tools.uniqify_sequence([pitches.WesternPitch(pitch_name) for pitch_name in 'c d e c d e e f a c a'.
→split(' ')])
[WesternPitch(c4),
```

(continues on next page)

(continued from previous page)

```
WesternPitch(d4),
WesternPitch(e4),
WesternPitch(f4),
WesternPitch(a4)]
```

cyclic_permutations(*sequence*)

Cyclic permutation of an iterable. Return a generator object.

Parameters

`sequence` (*Sequence[Any]*) – The sequence from which cyclic permutations shall be generated.

Return type

Generator

Example:

```
>>> from mutwo.utilities import tools
>>> permutations = tools.cyclic_permutations((1, 2, 3, 4))
>>> next(permutations)
(2, 3, 4, 1)
>>> next(permutations)
(3, 4, 1, 2)
```

Adapted function from the reply of Paritosh Singh

find_closest_index(*item*, *sequence*, *key=<function <lambda>>*)

Return index of element in `data` with smallest difference to `item`.

Parameters

- `item` (*Union[float, Fraction, int]*) – The item from which the closest item shall be found.
- `sequence` (*Sequence*) – The data to which the closest item shall be found.
- `key` (*Callable[[Any], T]*) –

Return type

int

Example:

```
>>> from mutwo.utilities import tools
>>> tools.find_closest_index(2, (1, 4, 5))
0
>>> tools.find_closest_index(127, (100, 4, 300, 53, 129))
4
>>> tools.find_closest_index(127, (('hi', 100), ('hey', 4), ('hello', 300)), key=lambda item: item[1])
0
```

find_closest_item(*item*, *sequence*, *key=<function <lambda>>*)

Return element in `data` with smallest difference to `item`.

Parameters

- `item` (*Union[float, Fraction, int]*) – The item from which the closest item shall be found.
- `sequence` (*Sequence*) – The data to which the closest item shall be found.
- `key` (*Callable[[Any], T]*) –

Returns

The closest number to `item` in `data`.

Return type

T

Example:

```
>>> from mutwo.utilities import tools
>>> tools.find_closest_item(2, (1, 4, 5))
1
>>> tools.find_closest_item(127, (100, 4, 300, 53, 129))
129
>>> tools.find_closest_item(
>>>     127,
>>>     (('hi', 100), ('hey', 4), ('hello', 300)),
>>>     key=lambda item: item[1]
>>> )
('hi', 100)
```

`get_nested_item_from_index_sequence`(*index_sequence*, *sequence*)

Get item in nested Sequence.

Parameters

- **index_sequence** (*Sequence[int]*) – The indices of the nested item.
- **sequence** (*Sequence[Any]*) – A nested sequence.

Return type

Any

Example:

```
>>> from mutwo.utilities import tools
>>> nested_sequence = (1, 2, (4, (5, 1), (9, (3,))))
>>> tools.get_nested_item_from_index_sequence((2, 2, 0), nested_sequence)
9
>>> nested_sequence[2][2][0]  # is equal
9
```

`set_nested_item_from_index_sequence`(*index_sequence*, *sequence*, *item*)

Set item in nested Sequence.

Parameters

- **index_sequence** (*Sequence[int]*) – The indices of the nested item which shall be set.
- **sequence** (*MutableSequence[Any]*) – A nested sequence.
- **item** (*Any*) – The new item value.

Return type

None

Example:

```
>>> from mutwo.utilities import tools
>>> nested_sequence = [1, 2, [4, [5, 1], [9, [3]]]]
>>> tools.set_nested_item_from_index_sequence((2, 2, 0), nested_sequence, 100)
>>> nested_sequence[2][2][0] = 100  # is equal
```

`find_numbers_which_sums_up_to`(*given_sum*, *number_to_choose_from_sequence=None*, *item_to_sum_up_count_set=None*)

Find all combinations of numbers which sum is equal to the given sum.

Parameters

- **given_sum** (*float*) – The target sum for which different combinations shall be searched.
- **number_to_choose_from_sequence** (*Optional[Sequence[float]]*) – A sequence of numbers which shall be tried to combine to result in the `given_sum`. If the user doesn't specify this argument mutwo will use all natural numbers equal or smaller than the `given_sum`.
- **item_to_sum_up_count_set** (*Optional[set[int]]*) – How many numbers can be combined to result in the `given_sum`. If the user doesn't specify this argument mutwo will use all natural numbers equal or smaller than the `given_sum`.

Return type

tuple[tuple[float, ...], ...]

Example:

```
>>> from mutwo.utilities import tools
>>> tools.find_numbers_which_sums_up_to(4)
((4,), (1, 3), (2, 2), (1, 1, 2), (1, 1, 1, 1))
```

call_function_except_attribute_error(*function*, *argument*, *exception_value*)

Run a function with argument as input

Parameters

- **function** (*Callable[[Any], Any]*) – The function to be called.
- **argument** (*Any*) – The argument with which the function shall be called.
- **exception_value** (*Any*) – The alternative value if the function call raises an *AttributeError*.

Returns

Return `exception_value` in case an attribute error occurs. In case the function call is successful the function return value will be returned.

Return type

Any

round_floats(*number_to_round*, *n_digits*)

Round number if it is an instance of float, otherwise unaltered number.

Parameters

- **number_to_round** (*core_constants.Real*) – The number which shall be rounded.
- **n_digits** (*int*) – How many digits shall the number be rounded.

Return type

Union[float, *Fraction*, int]

camel_case_to_snake_case(*camel_case_string*)

Transform camel case formatted string to snake case.

Parameters

camel_case_string (*str*) – String which is formatted using camel case (no whitespace, but upper letters at new word start).

Returns

string formatted using snake case

Return type

str

Example: MyClassName -> my_class_name

test_if_objects_are_equal_by_parameter_tuple(*object0*, *object1*, *parameter_to_compare_tuple*)

Check if the parameters of two objects have equal values.

Parameters

- **object0** (*Any*) – The first object which shall be compared.
- **object1** (*Any*) – The second object with which the first object shall be compared.
- **parameter_to_compare_tuple** (*tuple[str, ...]*) –

Parameter_to_compare_tuple

A tuple of attribute names which shall be compared.

Returns

True if all values of all parameters of the objects are equal and *False* if not or if an *AttributeError* is raised.

Return type

bool

Example:

```
>>> from mutwo import core_utilites
>>> class A: pass
>>> first_object = A()
>>> first_object.a = 100
```

(continues on next page)

(continued from previous page)

```
>>> second_object = A()
>>> second_object.a = 100
>>> third_object = A()
>>> third_object.a = 200
>>> core_utilites.test_if_objects_are_equal_by_parameter_tuple(
>>>     first_object, second_object, ("a",)
>>> )
True
>>> core_utilites.test_if_objects_are_equal_by_parameter_tuple(
>>>     first_object, third_object, ("a",)
>>> )
False
```

`get_all`(**submodule_tuple*)

Fetch from all arguments their *__all__* attribute and combine them to one tuple

Parameters
`submodule_tuple` (*module*) – Submodules which *__all__* attribute shall be fetched.

Return type
tuple[str, …]

This function is mostly useful in the *__init__* code of each *mutwo* module.

class `AlreadyDefinedValueNameError`(*cls*)

Bases: `Exception`

class `InvalidAverageValueStartAndEndWarning`

Bases: `RuntimeWarning`

class `InvalidStartValueError`(*start*, *duration*)

Bases: `Exception`

class `InvalidPointError`(*point*, *point_count*)

Bases: `Exception`

class `ImpossibleToSquashInError`(*event_to_be_squashed_into*, *event_to_squash_in*)

Bases: `TypeError`

class `InvalidStartAndEndValueError`(*start*, *end*)

Bases: `Exception`

class `InvalidCutOutStartAndEndValuesError`(*start*, *end*, *simple_event*, *duration*)

Bases: `Exception`

class `SplitUnavailableChildError`(*absolute_time*)

Bases: `Exception`

Parameters
`absolute_time` (*Union[float, Fraction, int]*) –

class `NoSolutionFoundError`(*message*)

Bases: `Exception`

Parameters
`message` (*str*) –

mutwo.core_utilities.configurations

Configure the default behaviour of utility functions

`PICKLE_MODULE_TO_SEARCH_TUPLE = ('cloudpickle', 'dill')`
    Define alternative pickle modules which are used in the `mutwo.core_utilites.compute_lazy()` decorator.

mutwo.core_version

Table of content

`VERSION = '0.61.7'`
    The version of the package `mutwo.core`.

mutwo.csound_converters

Table of content

Object	Documentation
mutwo.csound_converters.EventToCsoundScore	Class to convert mutwo events to a Csound score file.
mutwo.csound_converters.EventToSoundFile	Generate audio files with Csound.

`class EventToCsoundScore(**pfield)`

Bases: *EventConverter*

Class to convert mutwo events to a Csound score file.

Parameters
    pfield (*Callable[[SimpleEvent], Union[float, Fraction, int, str]]*) – p-field / p-field-extraction-function pairs.

This class helps generating score files for the "domain-specific computer programming language for audio programming" Csound.

EventToCsoundScore extracts data from mutwo Events and assign it to specific p-fields. The mapping of Event attributes to p-field values has to be defined by the user via keyword arguments during class initialization.

By default, mutwo already maps the following p-fields to the following values:

- p1 (instrument name) to 1
- p2 (start time) to the absolute start time of the event
- p3 (duration) to the `duration` attribute of the event

If p2 shall be assigned to the absolute entry delay of the event, it has to be set to None.

The *EventToCsoundScore* ignores any p-field that returns any unsupported p-field type (anything else than a string or a number). If the returned type is a string, *EventToCsoundScore* automatically adds quotations marks around the string in the score file.

All p-fields can be overwritten in the following manner:

```
>>> from mutwo import csound_converters
>>> my_converter = csound_converters.EventToCsoundScore(
>>>     p1=lambda event: 2,
>>>     p4=lambda event: event.pitch.frequency,
>>>     p5=lambda event: event.volume
>>> )
```

For easier debugging of faulty score files, *mutwo* adds annotations when a new `SequentialEvent` or a new `SimultaneousEvent` starts.

`convert`(*event_to_convert*, *path*)

Render csound score file (.sco) from the passed event.

Parameters

- **event_to_convert** (`core_events.abc.Event`) – The event that shall be rendered to a csound score file.
- **path** (*str*) – where to write the csound score file

Return type

None

```
>>> import random
>>> from mutwo import core_events
>>> from mutwo import csound_converters
>>> from mutwo import music_parameters
>>> converter = csound_converters.EventToCsoundScore(
>>>    p4=lambda event: event.pitch.frequency
>>> )
>>> events = core_events.SequentialEvent(
>>>    [
>>>        core_events.SimpleEvent(random.uniform(0.3, 1.2)) for _ in range(15)
>>>    ]
>>> )
>>> for event in events:
>>>     event.pitch = music_parameters.DirectPitch(random.uniform(100, 500))
>>> converter.convert(events, 'score.sco')
```

class EventToSoundFile(*csound_orchestra_path*, *event_to_csound_score*, **flag*, *remove_score_file=False*)

Bases: *Converter*

Generate audio files with Csound.

Parameters

- **csound_orchestra_path** (*str*) – Path to the csound orchestra (.orc) file.
- **event_to_csound_score** (`EventToCsoundScore`) – The *EventToCsoundScore* that shall be used to render the csound score file (.sco) from a mutwo event.
- ***flag** (*str*) – Flag that shall be added when calling csound. Several of the supported csound flags can be found in *mutwo.csound_converters.constants*.
- **remove_score_file** (*bool*) – Set to True if *EventToSoundFile* shall remove the csound score file after rendering. Defaults to False.

Disclaimer: Before using the *EventToSoundFile*, make sure Csound has been correctly installed on your system.

`convert`(*event_to_convert*, *path*, *score_path=None*)

Render sound file from the mutwo event.

Parameters

- **event_to_convert** (`core_events.abc.Event`) – The event that shall be rendered.
- **path** (*str*) – where to write the sound file
- **score_path** (*Optional[str]*) – where to write the score file

Return type

None

mutwo.csound_converters.configurations

Configure the behaviour of *mutwo.csound_converters*.

`N_EMPTY_LINES_AFTER_COMPLEX_EVENT = 1`

How many empty lines shall be written to a Csound Score file after a `ComplexEvent`.

`SEQUENTIAL_EVENT_ANNOTATION = ';; NEW SEQUENTIAL EVENT\n;;'`

Annotation in Csound Score files when a new `SequentialEvent` starts.

`SIMULTANEOUS_EVENT_ANNOTATION = ';; NEW SIMULTANEOUS EVENT\n;;'`

Annotation in Csound Score files when a new `SimultaneousEvent` starts.

mutwo.csound_converters.constants

Constants to be used for and with *mutwo.csound_converters*.

The file mostly contains different flags for running Csound. The flag definitions are documented here.

`FORMAT_24BIT = '--format=24bit'`

Flag for rendering sound files in 24bit.

`FORMAT_64BIT = '--format=double'`

Flag for rendering sound files in 64bit floating point.

`FORMAT_8BIT = '--format=uchar'`

Flag for rendering sound files in 8bit.

`FORMAT_FLOAT = '--format=float'`

Flag for rendering sound files in single-format float audio samples.

`FORMAT_IRCAM = '--format=ircam'`

Flag for rendering sound files in IRCAM format.

`FORMAT_WAV = '--format=wav'`

Flag for rendering sound files in wav file format.

`SILENT_FLAG = '-O null'`

Flag for preventing Csound from printing any information while rendering.

mutwo.csound_version

Table of content

`VERSION = '0.6.1'`

The version of the package `mutwo.csound`.

mutwo.ekmelily_converters

Table of content

Object	Documentation
mutwo.ekmelily_converters.EkmelilyAccidental	Representation of an Ekmelily accidental.
mutwo.ekmelily_converters.EkmelilyTuningFileConverter	Build Ekmelily tuning files from Ekmelily accidentals.
mutwo.ekmelily_converters.HEJIEkmelilyTuningFileConverter	Build Ekmelily tuning files for Helmholtz-Ellis JI Pitch Notation.

class EkmelilyAccidental(*accidental_name*, *accidental_glyph_tuple*, *deviation_in_cents*, *available_diatonic_pitch_index_tuple=None*)

Bases: `object`

Representation of an Ekmelily accidental.

Parameters

- **accidental_name** (*str*) – The name of the accidental that follows after the diatonic pitch name (e.g. 's' or 'qf')
- **accidental_glyph_tuple** (*tuple[str, ...]*) – The name of accidental glyphs that should appear before the notehead. For a list of available glyphs, check the documentation of Ekmelos. Furthermore one can find mappings from mutwo data to Ekmelos glyph names in `PRIME_AND_EXPONENT_AND_TRADITIONAL_ACCIDENTAL_TO_ACCIDENTAL_GLYPH_DICT` and `TEMPERED_ACCIDENTAL_TO_ACCIDENTAL_GLYPH_DICT`.
- **deviation_in_cents** (*float*) – How many cents shall an altered pitch differ from its diatonic / natural counterpart.
- **available_diatonic_pitch_index_tuple** (*Optional[tuple[int, ...]], optional*) – Sometimes one may want to define accidentals which are only available for certain diatonic music_parameters. For this case, one can use this argument and specify all diatonic music_parameters which should know this accidental. If this argument keeps undefined, the accidental will be added to all seven diatonic music_parameters.

Example:

```
>>> from mutwo.ext.converter.frontends import ekmelily
>>> natural = ekmelily.EkmelilyAccidental('', ("#xE261",), 0)
>>> sharp = ekmelily.EkmelilyAccidental('s', ("#xE262",), 100)
>>> flat = ekmelily.EkmelilyAccidental('f', ("#xE260",), -100)
```

accidental_glyph_tuple: tuple[str, ...]

accidental_name: str

available_diatonic_pitch_index_tuple: Optional[tuple[int, ...]] = None

deviation_in_cents: float

class EkmelilyTuningFileConverter(*path*, *ekmelily_accidental_sequence*, *global_scale=None*)

Bases: *Converter*

Build Ekmelily tuning files from Ekmelily accidentals.

Parameters

- **path** (*str*) – Path where the new Ekmelily tuning file shall be written. The suffix '.ily' is recommended, but not necessary.
- **ekmelily_accidental_sequence** (*Sequence[EkmelilyAccidental]*) – A sequence which contains all *EkmelilyAccidental* that shall be written to the tuning file,
- **global_scale** (*tuple[fractions.Fraction, ...], optional*) – From the Lilypond documentation: "This determines the tuning of music_parameters with no accidentals or key signatures. The first pitch is c. Alterations are calculated relative to this scale. The number of music_parameters in this scale determines the number of scale steps that make up an octave. Usually the 7-note major scale."

Example:

```
>>> from mutwo.converter.frontends import ekmelily
>>> natural = ekmelily.EkmelilyAccidental('', ("#xE261",), 0)
>>> sharp = ekmelily.EkmelilyAccidental('s', ("#xE262",), 100)
>>> flat = ekmelily.EkmelilyAccidental('f', ("#xE260",), -100)
>>> eigth_tone_sharp = ekmelily.EkmelilyAccidental('es', ("#xE2C7",), 25)
>>> eigth_tone_flat = ekmelily.EkmelilyAccidental('ef', ("#xE2C2",), -25)
>>> converter = ekmelily.EkmelilyTuningFileConverter(
>>>     'ekme-test.ily', (natural, sharp, flat, eigth_tone_sharp, eigth_tone_flat)
>>> )
>>> converter.convert()
```

`convert()`

Render tuning file to `path`.

`class HEJIEkmelilyTuningFileConverter`(*path=None*, *prime_to_highest_allowed_exponent=None*, *reference_pitch='c'*, *prime_to_heji_accidental_name=None*, *otonality_indicator=None*, *utonality_indicator=None*, *exponent_to_exponent_indicator=None*, *tempered_pitch_indicator=None*, *set_microtonal_tuning=True*)

Bases: *EkmelilyTuningFileConverter*

Build Ekmelily tuning files for Helmholtz-Ellis JI Pitch Notation.

Parameters

- **path** (*str*) – Path where the new Ekmelily tuning file shall be written. The suffix '.ily' is recommended, but not necessary.
- **prime_to_highest_allowed_exponent** (*dict[int, int], optional*) – Mapping of prime number to highest exponent that should occur. Take care not to add higher exponents than the HEJI Notation supports. See *DEFAULT_PRIME_TO_HIGHEST_ALLOWED_EXPONENT_DICT* for the default mapping.
- **reference_pitch** (*str, optional*) – The reference pitch (1/1). Should be a diatonic pitch name (see `DIATONIC_PITCH_CLASS_CONTAINER`) in English nomenclature. For any other reference pitch than 'c', Lilyponds midi rendering for music_parameters with the diatonic pitch 'c' will be slightly out of tune (because the first value of *global_scale* always have to be 0).
- **prime_to_heji_accidental_name** (*dict[int, str], optional*) – Mapping of a prime number to a string which indicates the respective prime number in the resulting accidental name. See *DEFAULT_PRIME_TO_HEJI_ACCIDENTAL_NAME_DICT* for the default mapping.
- **otonality_indicator** (*str, optional*) – String which indicates that the respective prime alteration is otonal. See *DEFAULT_OTONALITY_INDICATOR* for the default value.
- **utonality_indicator** (*str, optional*) – String which indicates that the respective prime alteration is utonal. See *DEFAULT_OTONALITY_INDICATOR* for the default value.
- **exponent_to_exponent_indicator** (*Callable[[int], str], optional*) – Function to convert the exponent of a prime number to string which indicates the respective exponent. See *DEFAULT_EXPONENT_TO_EXPONENT_INDICATOR()* for the default function.
- **tempered_pitch_indicator** (*str, optional*) – String which indicates that the respective accidental is tempered (12 EDO). See *DEFAULT_TEMPERED_PITCH_INDICATOR* for the default value.
- **set_microtonal_tuning** (*bool*) – If set to `False` the converter won't apply any microtonal music_parameters. In this case all chromatic music_parameters will return normal 12EDO music_parameters. Default to `True`.

mutwo.ekmelily_converters.configurations

Configure default behaviour of *mutwo.ekmelily_converters*

`DEFAULT_EXPONENT_TO_EXPONENT_INDICATOR`(*exponent*)

Default function for `HEJIEkmelilyTuningFileConverter` argument *exponent_to_exponent_indicator*.

`DEFAULT_GLOBAL_SCALE = (Fraction(0, 1), Fraction(1, 1), Fraction(2, 1), Fraction(5, 2), Fraction(7, 2), Fraction(9, 2), Fraction(11, 2))`

Default value for `EkmelilyTuningFileConverter` argument *global_scale*.

`DEFAULT_OTONALITY_INDICATOR = 'o'`

Default value for `HEJIEkmelilyTuningFileConverter` argument *otonality_indicator*.

`DEFAULT_PRIME_TO_HEJI_ACCIDENTAL_NAME_DICT = {5: 'a', 7: 'b', 11: 'c', 13: 'd', 17: 'e', 19: 'f', 23: 'g'}`

Default mapping for `HEJIEkmelilyTuningFileConverter` argument *prime_to_heji_accidental_name*.

`DEFAULT_PRIME_TO_HIGHEST_ALLOWED_EXPONENT_DICT = {5: 3, 7: 2, 11: 1, 13: 1, 17: 1}`

Default value for `HEJIEkmelilyTuningFileConverter` argument *prime_to_highest_allowed_exponent*.

`DEFAULT_TEMPERED_PITCH_INDICATOR = 't'`

Default value for `HEJIEkmelilyTuningFileConverter` argument *tempered_pitch_indicator*.

`DEFAULT_UTONALITY_INDICATOR = 'u'`

Default value for `HEJIEkmelilyTuningFileConverter` argument *utonality_indicator*.

mutwo.ekmelily_converters.constants

Constants to be used for and with *mutwo.ekmelily_converters*.

`DIFFERENCE_BETWEEN_PYTHAGOREAN_AND_TEMPERED_FIFTH = 1.955000865387433`

The difference in cents between a just fifth (3/2) and a 12-EDO fifth. This constant is used in `HEJIEkmelilyTuningFileConverter`.

`PRIME_AND_EXPONENT_AND_TRADITIONAL_ACCIDENTAL_TO_ACCIDENTAL_GLYPH_DICT = {(None, None, ''): '#xE261', (None, None, 's'): '#xE262', (None, None, 'ss'): '#xE263', (None, None, 'f'): '#xE260', (None, None, 'ff'): '#xE264', (5, 1, ''): '#xE2C2', (5, 2, ''): '#xE2C2', (5, 3, ''): '#xE2D6', (5, -1, ''): '#xE2C7', (5, -2, ''): '#xE2D1', (5, -3, ''): '#xE2DB', (5, 1, 's'): '#xE2C3', (5, 2, 's'): '#xE2CD', (5, 3, 's'): '#xE2D7', (5, -1, 's'): '#xE2C8', (5, -2, 's'): '#xE2D2', (5, -3, 's'): '#xE2DC', (5, 1, 'ss'): '#xE2C4', (5, 2, 'ss'): '#xE2CE', (5, 3, 'ss'): '#xE2D8', (5, -1, 'ss'): '#xE2C9', (5, -2, 'ss'): '#xE2D3', (5, -3, 'ss'): '#xE2DD', (5, 1, 'f'): '#xE2C1', (5, 2, 'f'): '#xE2CB', (5, 3, 'f'): '#xE2D5', (5, -1, 'f'): '#xE2C6', (5, -2, 'f'): '#xE2D0', (5, -3, 'f'): '#xE2DA', (5, 1, 'ff'): '#xE2C0', (5, 2, 'ff'): '#xE2CA', (5, 3, 'ff'): '#xE2D4', (5, -1, 'ff'): '#xE2C5', (5, -2, 'ff'): '#xE2CF', (5, -3, 'ff'): '#xE2D9', (7, 1, None): '#xE2DE', (7, 2, None): '#xE2E0', (7, -1, None): '#xE2DF', (7, -2, None): '#xE2E1', (11, 1, None): '#xE2E3', (11, -1, None): '#xE2E2', (13, 1, None): '#xE2E4', (13, -1, None): '#xE2E5', (17, 1, None): '#xE2E6', (17, -1, None): '#xE2E7', (19, 1, None): '#xE2E9', (19, -1, None): '#xE2E8', (23, 1, None): '#xE2EA', (23, -1, None): '#xE2EB'}`

Mapping of prime, exponent and pythagorean accidental to accidental glyph name in Ekmelos.

`PYTHAGOREAN_ACCIDENTAL_CENT_DEVIATION_SIZE = 113.69`

Step in cents for one pythagorean accidental (# or b).

`PYTHAGOREAN_ACCIDENTAL_TO_CENT_DEVIATION_DICT = {'': 0, 'f': -113.69, 'ff': -227.38, 's': 113.69, 'ss': 227.38}`

Step in cents mapping for each pythagorean accidental (# or b).

`TEMPERED_ACCIDENTAL_TO_ACCIDENTAL_GLYPH_DICT = {'': '#xE2F2', 'f': '#xE2F1', 'ff': '#xE2F0', 'qf': '#xE2F5', 'qs': '#xE2F6', 's': '#xE2F3', 'ss': '#xE2F4'}`

Mapping of tempered accidental name to glyph name in Ekmelos.

`TEMPERED_ACCIDENTAL_TO_CENT_DEVIATION_DICT = {'': 0, 'f': -100, 'ff': -200, 'qf': -50, 'qs': 50, 's': 100, 'ss': 200}`

Mapping of tempered accidental name to cent deviation.

mutwo.ekmelily_version

Table of content

`VERSION = '0.7.2'`

The version of the package `mutwo.ekmelily`.

mutwo.isis_converters

Table of content

Object	Documentation
mutwo.isis_converters.EventToIsisScore	Class to convert mutwo events to a ISiS score file.
mutwo.isis_converters.EventToSingingSynthesis	Generate audio files with ISiS.

class EventToIsisScore(*simple_event_to_pitch=<function EventToIsisScore.<lambda>>, simple_event_to_volume=<function EventToIsisScore.<lambda>>, simple_event_to_vowel=<function EventToIsisScore.<lambda>>, simple_event_to_consonant_tuple=<function EventToIsisScore.<lambda>>, is_simple_event_rest=<function EventToIsisScore.<lambda>>, tempo=60, global_transposition=0, default_sentence_loudness=None, n_events_per_line=5*)

Bases: *EventConverter*

Class to convert mutwo events to a ISiS score file.

Parameters

- **simple_event_to_pitch** (*Callable[[SimpleEvent], Pitch]*) – Function to extract an instance of *mutwo.music_parameters.abc.Pitch* from a simple event.
- **simple_event_to_volume** (*Callable[[SimpleEvent], Volume]*) –
- **simple_event_to_vowel** (*Callable[[SimpleEvent], str]*) –
- **simple_event_to_consonant_tuple** (*Callable[[SimpleEvent], tuple[str, ...]]*) –
- **is_simple_event_rest** (*Callable[[SimpleEvent], bool]*) –
- **tempo** (*Union[float, Fraction, int]*) – Tempo in beats per minute (BPM). Defaults to 60.
- **global_transposition** (*int*) – global transposition in midi numbers. Defaults to 0.
- **n_events_per_line** (*int*) – How many events the score shall contain per line. Defaults to 5.
- **default_sentence_loudness** (*Optional[Union[float, Fraction, int]]*) –

convert(*event_to_convert, path*)

Render ISiS score file from the passed event.

Parameters

- **event_to_convert** (*Union[core_events.SimpleEvent, core_events.SequentialEvent[core_events.SimpleEvent]]*) – The event that shall be rendered to a ISiS score file.
- **path** (*str*) – where to write the ISiS score file

Return type

None

Example:

```
>>> from mutwo import core_events
>>> from mutwo import music_events
>>> from mutwo import music_parameters
>>> from mutwo import isis_converters
>>> notes = core_events.SequentialEvent(
>>>     [
>>>          music_events.NoteLike(music_parameters.WesternPitch(pitch_name), 0.5, 0.5)
>>>          for pitch_name in 'c f d g'.split(' ')
>>>     ]
>>> )
>>> for consonants, vowel, note in zip([[], [], ['t'], []], ['a', 'o', 'e', 'a'], notes):
>>>     note.vowel = vowel
>>>     note.consonants = consonants
>>> event_to_isis_score = isis.EventToIsisScore('my_singing_score')
>>> event_to_isis_score.convert(notes)
```

class EventToSingingSynthesis(*isis_score_converter, *flag, remove_score_file=False, isis_executable_path=None*)

Bases: *Converter*

Generate audio files with ISiS.

Parameters

- **isis_score_converter** (EventToIsisScore) – The *EventToIsisScore* that shall be used to render the ISiS score file from a mutwo event.
- ***flag** (*str*) – Flag that shall be added when calling ISiS. Several of the supported ISiS flags can be found in *mutwo.isis_converters.constants*.
- **remove_score_file** (*bool*) – Set to True if *EventToSingingSynthesis* shall remove the ISiS score file after rendering. Defaults to False.

- **isis_executable_path** (*Optional[str]*) – The path to the ISiS executable (binary file). If not specified the value of *mutwo.isis_converters.configurations.DEFAULT_ISIS_EXECUTABLE_PATH* will be used.

Disclaimer: Before using the *EventToSingingSynthesis*, make sure ISiS has been correctly installed on your system.

`convert`(*event_to_convert*, *path*, *score_path=None*)

Render sound file via ISiS from mutwo event.

Parameters

- **event_to_convert** (*Union[SimpleEvent, SequentialEvent[SimpleEvent]]*) – The event that shall be rendered.
- **path** (*str*) – The path / filename of the resulting sound file
- **score_path** (*Optional[str]*) – The path where the score file shall be written to.

Return type

None

Disclaimer: Before using the *EventToSingingSynthesis*, make sure ISiS has been correctly installed on your system.

mutwo.isis_converters.configurations

Configure the behaviour of classes in *mutwo.isis_converters*

`DEFAULT_ISIS_EXECUTABLE_PATH = 'isis.sh'`

The path to the ISiS shell script. When installing ISiS with the packed 'Install_ISiS_commandline.sh' script, the path should be 'isis.sh'.

mutwo.isis_converters.constants

Constants to be used for and with *mutwo.isis_converters*.

The file mostly contains different flags for running ISiS. The flag definitions are documented here.

`SECTION_LYRIC_NAME = 'lyrics'`

Section name for lyrics in score config file

`SECTION_SCORE_NAME = 'score'`

Section name for score in score config file

`SILENT_FLAG = '--quiet'`

Flag for preventing ISiS from printing any information during rendering.

mutwo.isis_utilities

Table of content

Object	Documentation
mutwo.isis_utilities.MonophonicSynthesizerError	

`class MonophonicSynthesizerError`

Bases: `Exception`

mutwo.isis_version

Table of content

`VERSION = '0.8.2'`

The version of the package `mutwo.isis`.

mutwo.mbrola_converters

Table of content

Object	Documentation
mutwo.mbrola_converters.EventToPhonemeList	Convert mutwo event to `voxpopuli.PhonemeList`.
mutwo.mbrola_converters.EventToSpeakSynthesis	Render event to soundfile with speak synthesis engine mbrola.
mutwo.mbrola_converters.SimpleEventToPitch	Convert a simple event to a pitch.
mutwo.mbrola_converters.SimpleEventToPhonemeString	Convert a simple event to a phoneme string.

`class EventToPhonemeList`(*simple_event_to_pitch=<mutwo.mbrola_converters.mbrola.SimpleEventToPitch object>, simple_event_to_phoneme_string=<mutwo.mbrola_converters.mbrola.SimpleEventToPhonemeString object>*)

Bases: *EventConverter*

Convert mutwo event to `voxpopuli.PhonemeList`.

Parameters

- **simple_event_to_pitch** (*Callable[[core_events.SimpleEvent], Optional[music_parameters.abc.Pitch]]*) – Function or converter which receives a *mutwo.core_events.SimpleEvent* as an input and has to return a :class`mutwo.music_parameters.abc.Pitch` or *None*.
- **simple_event_to_phoneme_string** (*Callable[[core_events.SimpleEvent], str]*) – Function or converter which receives a *mutwo.core_events.SimpleEvent* as an input and has to return a string which belongs to the phonetic alphabet SAMPA.

Warning:

This converter assumes that the duration attribute of the input event is in seconds. It multiplies the input duration by a factor of 1000 and parses it to the *voxpopuli.Phoneme* object which expects duration in milliseconds. It is the responsibility of the user to ensure that the duration has the right format.

`convert`(*event_to_convert*)

Parameters

event_to_convert (Event) –

Return type

PhonemeList

`class EventToSpeakSynthesis`(*voice=<voxpopuli.main.Voice object>, event_to_phoneme_list=<mutwo.mbrola_converters.mbrola.EventToPhonemeList object>*)

Bases: *Converter*

Render event to soundfile with speak synthesis engine mbrola.

Parameters

- **voice** (*voxpopuli.Voice*) – The voice object which is responsible in rendering the soundfile.
- **event_to_phoneme_list** (*Callable[[core_events.abc.Event], voxpopuli.PhonemeList]*) – A converter or function which transforms an event to a `voxpopuli.PhonemeList`. By default this is a *mutwo.mbrola_converters.EventToPhonemeList* object..

Warning:

You need to install the non-python dependencies for *voxpopuli*, otherwise the converter won't work.

`convert`(*event_to_convert*, *sound_file_name*)

Parameters

- **event_to_convert** (Event) –
- **sound_file_name** (*str*) –

class **SimpleEventToPitch**(*attribute_name=None*, *exception_value=[]*)

Bases: *SimpleEventToPitchList*

Convert a simple event to a pitch.

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (*list[*mutwo.music_parameters.abc.Pitch*]*) –

`convert`(**args*, ***kwargs*)

Extract from a *mutwo.core_events.SimpleEvent* an attribute.

Parameters

simple_event_to_convert (mutwo.core_events.SimpleEvent) – The *mutwo.core_events.SimpleEvent* from which an attribute shall be extracted.

Return type

Optional[Pitch]

Example:

```
>>> from mutwo import core_converters
>>> from mutwo import core_events
>>> simple_event = core_events.SimpleEvent(duration=10)
>>> simple_event_to_duration = core_converters.SimpleEventToAttribute(
        'duration', 0
    )
>>> simple_event_to_duration.convert(simple_event)
10
>>> simple_event_to_pasta = core_converters.SimpleEventToAttribute(
        'pasta', 'spaghetti'
    )
>>> simple_event_to_pasta.convert(simple_event)
'spaghetti'
>>> simple_event.pasta = 'tagliatelle'
>>> simple_event_to_pasta.convert(simple_event)
'tagliatelle'
```

class **SimpleEventToPhonemeString**(*attribute_name='phoneme'*, *exception_value='_'*)

Bases: *SimpleEventToAttribute*

Convert a simple event to a phoneme string.

Parameters

- **attribute_name** (*str*) –
- **exception_value** (*str*) –

mutwo.mbrola_version

Table of content

`VERSION = '0.3.1'`

The version of the package `mutwo.mbrola`.

mutwo.midi_converters

Table of content

Object	Documentation
mutwo.midi_converters.PitchBendingNumberToPitchInterval	Convert midi pitch bend number to *mutwo.music_parameters.abc.PitchInterval*.
mutwo.midi_converters.PitchBendingNumberToDirectPitchInterval	Convert midi pitch bend number to *mutwo.music_parameters.DirectPitchInterval*.
mutwo.midi_converters.MidiPitchToMutwoPitch	Convert midi pitch to *mutwo.music_parameters.abc.Pitch*.
mutwo.midi_converters.MidiPitchToDirectPitch	
mutwo.midi_converters.MidiPitchToMutwoMidiPitch	
mutwo.midi_converters.MidiVelocityToMutwoVolume	Convert midi velocity (integer) to *mutwo.music_parameters.abc.Volume*.
mutwo.midi_converters.MidiVelocityToWesternVolume	
mutwo.midi_converters.MidiFileToEvent	Convert a midi file to a mutwo event.
mutwo.midi_converters.SimpleEventToControlMessageTuple	Convert *mutwo.core_events.SimpleEvent* to a tuple of control messages
mutwo.midi_converters.CentDeviationToPitchBendingNumber	Convert cent deviation to midi pitch bend number.
mutwo.midi_converters.MutwoPitchToMidiPitch	Convert mutwo pitch to midi pitch number and midi pitch bend number.
mutwo.midi_converters.EventToMidiFile	Class for rendering standard midi files (SMF) from mutwo data.

class PitchBendingNumberToPitchInterval(*maximum_pitch_bend_deviation=None*)

Bases: *Converter*

Convert midi pitch bend number to *mutwo.music_parameters.abc.PitchInterval*.

Parameters

maximum_pitch_bend_deviation (*int*) – sets the maximum pitch bending range in cents. This value depends on the particular used software synthesizer and its settings, because it is up to the respective synthesizer how to interpret the pitch bending messages. By default mutwo sets the value to 200 cents which seems to be the most common interpretation among different manufacturers.

abstract convert(*pitch_bending_number_to_convert*)

Parameters

pitch_bending_number_to_convert (*int*) –

Return type

PitchInterval

class PitchBendingNumberToDirectPitchInterval(*maximum_pitch_bend_deviation=None*)

Bases: *PitchBendingNumberToPitchInterval*

Convert midi pitch bend number to *mutwo.music_parameters.DirectPitchInterval*.

Parameters

maximum_pitch_bend_deviation (*Optional[float]*) –

convert(*pitch_bending_number_to_convert*)

Convert pitch bending number to *mutwo.music_parameters.DirectPitchInterval*

Parameters

pitch_bending_number_to_convert (*midi_converters.constants.PitchBend*) – The pitch bending number which shall be converted.

Return type

DirectPitchInterval

class MidiPitchToMutwoPitch(*pitch_bending_number_to_pitch_interval=<mutwo.midi_converters.backends.PitchBendingNumberToDirectPitchInterval object>*)

Bases: *Converter*

Convert midi pitch to *mutwo.music_parameters.abc.Pitch*.

Parameters

pitch_bending_number_to_pitch_interval (*Callable[[midi_converters.constants.PitchBend], music_parameters.abc.PitchInterval]*) – A callable object which transforms a pitch bending number (integer) to a *mutwo.music_parameters.abc.PitchInterval*. Default to *PitchBendingNumberToDirectPitchInterval*.

abstract convert(*midi_pitch_to_convert*)

Parameters

midi_pitch_to_convert (*tuple[int, int]*) –

Return type

Pitch

class MidiPitchToDirectPitch(*pitch_bending_number_to_pitch_interval=<mutwo.midi_converters.backends.PitchBendingNumberToDirectPitchInterval object>*)

Bases: *MidiPitchToMutwoPitch*

Parameters

pitch_bending_number_to_pitch_interval (*Callable[[int], PitchInterval]*) –

convert(*midi_pitch_to_convert*)

Parameters

midi_pitch_to_convert (*tuple[int, int]*) –

Return type

DirectPitch

class MidiPitchToMutwoMidiPitch(*pitch_bending_number_to_pitch_interval=<mutwo.midi_converters.backends.PitchBendingNumberToDirectPitchInterval object>*)

Bases: *MidiPitchToMutwoPitch*

Parameters

pitch_bending_number_to_pitch_interval (*Callable[[int], PitchInterval]*) –

convert(*midi_pitch_to_convert*)

Parameters

midi_pitch_to_convert (*tuple[int, int]*) –

Return type

MidiPitch

class MidiVelocityToMutwoVolume

Bases: *Converter*

Convert midi velocity (integer) to *mutwo.music_parameters.abc.Volume*.

abstract convert(*midi_velocity*)

Parameters

midi_velocity (*int*) –

Return type

Volume

class MidiVelocityToWesternVolume

Bases: *MidiVelocityToMutwoVolume*

convert(*midi_velocity_to_convert*)

Convert midi velocity to *mutwo.music_parameters.WesternVolume*

Parameters

midi_velocity_to_convert (*midi_converters.constants.MidiVelocity*) – The velocity which shall be converted.

Return type

Volume

Example:

```
>>> from mutwo import midi_converters
>>> midi_converters.MidiVelocityToWesternVolume().convert(127)
WesternVolume(fffff)
>>> midi_converters.MidiVelocityToWesternVolume().convert(0)
WesternVolume(ppppp)
```

class MidiFileToEvent(*mutwo_parameter_dict_to_simple_event=<mutwo.music_converters.parsers.MutwoParameterDictToNoteLike object>, midi_pitch_to_mutwo_pitch=<mutwo.midi_converters.backends.MidiPitchToMutwoMidiPitch object>, midi_velocity_to_mutwo_volume=<mutwo.midi_converters.backends.MidiVelocityToWesternVolume object>*)

Bases: *Converter*

Convert a midi file to a mutwo event.

Parameters

- **mutwo_parameter_tuple_to_simple_event** (*Callable[[tuple[core_constants.DurationType, music_parameters.abc.Pitch, music_parameters.abc.Volume]], core_events.SimpleEvent]*) – A callable which converts a tuple of mutwo parameters (duration, pitch list, volume) to a *mutwo.core_events.SimpleEvent*. In default state mutwo

  generates a *mutwo.music_events.NoteLike*.

- **midi_pitch_to_mutwo_pitch** (*Callable[[midi_converters.constants.MidiPitch], music_parameters.abc.Pitch]*) – Callable object which converts midi pitch (integer) to a *mutwo.music_parameters.abc.Pitch*. Default to *MidiPitchToMutwoMidiPitch*.
- **midi_velocity_to_mutwo_volume** (*Callable[[midi_converters.constants.MidiVelocity], music_parameters.abc.Volume]*) – Callable object which converts midi velocity (integer) to a `mutwo.music_parameters.abc.Voume`. Default to `MidiPitchToWesternVolume`.
- **mutwo_parameter_dict_to_simple_event** (*Callable[[dict[str, Any]], SimpleEvent]*) –

Warning:

This is an unstable early version of the converter. Expect bugs when using it!

Disclaimer:

This conversion is incomplete: Not all information from a midi file will be used. In its current state the converter only takes into account midi notes (pitch, velocity and duration) and ignores all other midi messages.

convert(*midi_file_path_or_mido_midi_file*)

Convert midi file to mutwo event.

Parameters

midi_file_path_or_mido_midi_file (*Union[str, mido.MidiFile]*) – The midi file which shall be converted. Can either be a file path or a `MidiFile` object from the mido package.

Return type

Event

class SimpleEventToControlMessageTuple(*attribute_name=None, exception_value=()*)

Bases: *SimpleEventToAttribute*

Convert *mutwo.core_events.SimpleEvent* to a tuple of control messages

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (*tuple[mido.messages.messages.Message, ...]*) –

class `CentDeviationToPitchBendingNumber`(*maximum_pitch_bend_deviation=None*)

Bases: *Converter*

Convert cent deviation to midi pitch bend number.

Parameters
maximum_pitch_bend_deviation (*int*) – sets the maximum pitch bending range in cents. This value depends on the particular used software synthesizer and its settings, because it is up to the respective synthesizer how to interpret the pitch bending messages. By default mutwo sets the value to 200 cents which seems to be the most common interpretation among different manufacturers.

`convert`(*cent_deviation*)

Parameters
cent_deviation (*Union[float, Fraction, int]*) –

Return type
int

class `MutwoPitchToMidiPitch`(*cent_deviation_to_pitch_bending_number=<mutwo.midi_converters.frontends.CentDeviationToPitchBendingNumber object>*)

Bases: *Converter*

Convert mutwo pitch to midi pitch number and midi pitch bend number.

Parameters

- **maximum_pitch_bend_deviation** (*int*) – sets the maximum pitch bending range in cents. This value depends on the particular used software synthesizer and its settings, because it is up to the respective synthesizer how to interpret the pitch bending messages. By default mutwo sets the value to 200 cents which seems to be the most common interpretation among different manufacturers.
- **cent_deviation_to_pitch_bending_number** (CentDeviationToPitchBendingNumber) –

`convert`(*mutwo_pitch_to_convert*, *midi_note=None*)

Find midi note and pitch bending for given mutwo pitch

Parameters

- **mutwo_pitch_to_convert** (`music_parameters.abc.Pitch`) – The mutwo pitch which shall be converted.
- **midi_note** (*Optional[int]*) – Can be set to a midi note value if one wants to force the converter to calculate the pitch bending deviation for the passed midi note. If this argument is `None` the converter will simply use the closest midi pitch number to the passed mutwo pitch. Default to `None`.

Return type
tuple[int, int]

class `EventToMidiFile`(*simple_event_to_pitch_list=<mutwo.music_converters.parsers.SimpleEventToPitchList object>, simple_event_to_volume=<mutwo.music_converters.parsers.SimpleEventToVolume object>, simple_event_to_control_message_tuple=<mutwo.midi_converters.frontends.SimpleEventToControlMessageTuple object>, midi_file_type=None, available_midi_channel_tuple=None, distribute_midi_channels=False, n_midi_channels_per_track=None, mutwo_pitch_to_midi_pitch=<mutwo.midi_converters.frontends.MutwoPitchToMidiPitch object>, ticks_per_beat=None, instrument_name=None, tempo_envelope=None*)

Bases: *Converter*

Class for rendering standard midi files (SMF) from mutwo data.

Mutwo offers a wide range of options how the respective midi file shall be rendered and how mutwo data shall be translated. This is necessary due to the limited and not always unambiguous nature of musical encodings in midi files. In this way the user can tweak the conversion routine to her or his individual needs.

Parameters

- **simple_event_to_pitch_list** (*Callable[[*`core_events.SimpleEvent`*], tuple[*`music_parameters.abc.Pitch`*, ...]]*) – Function to extract from a *`mutwo.core_events.SimpleEvent`* a tuple that contains pitch objects (objects that inherit from `mutwo.ext.parameters.abc.Pitch`). By default it asks the Event for its `pitch_list` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their pitch property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no pitch can be extracted), mutwo will interpret the event as a rest.
- **simple_event_to_volume** (*Callable[[*`core_events.SimpleEvent`*],* `music_parameters.abc.Volume`*]*) – Function to extract the volume from a *`mutwo.core_events.SimpleEvent`* in the purpose of generating midi notes. The function should return an object that inhertis from `mutwo.ext.parameters.abc.Volume`. By default it asks the Event for

its `volume` attribute (because by default `mutwo.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their volume property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no volume can be extracted), mutwo will interpret the event as a rest.

- **`simple_event_to_control_message_tuple`** (*`Callable[ [core_events.SimpleEvent], tuple[mido.Message, ...]]`*) – Function to generate midi control messages from a simple event. By default no control messages are generated. If the function call raises an AttributeError (e.g. if an expected control value isn't available) mutwo will interpret the event as a rest.
- **`midi_file_type`** (*`int`*) – Can either be 0 (for one-track midi files) or 1 (for synchronous multi-track midi files). Mutwo doesn't offer support for generating type 2 midi files (midi files with asynchronous tracks).
- **`available_midi_channel_tuple`** (*`tuple[int, ...]`*) – tuple containing integer where each integer represents the number of the used midi channel. Integer can range from 0 to 15. Higher numbers of available_midi_channel_tuple (like all 16) are recommended when rendering microtonal music. It shall be remarked that midi-channel 9 (or midi channel 10 when starting to count from 1) is often ignored by several software synthesizer, because this channel is reserved for percussion instruments.
- **`distribute_midi_channels`** (*`bool`*) – This parameter is only relevant if more than one *`SequentialEvent`* is passed to the convert method. If set to `True` each *`SequentialEvent`* only makes use of exactly n_midi_channel (see next parameter). If set to `False` each converted `SequentialEvent` is allowed to make use of all available channels. If set to `True` and the amount of necessary MidiTracks is higher than the amount of available channels, mutwo will silently cycle through the list of available midi channel.
- **`n_midi_channels_per_track`** (*`int`*) – This parameter is only relevant for distribute_midi_channels == True. It sets how many midi channels are assigned to one SequentialEvent. If microtonal chords shall be played by one SequentialEvent (via pitch bending messages) a higher number than 1 is recommended. Defaults to 1.
- **`mutwo_pitch_to_midi_pitch`** (*`MutwoPitchToMidiPitch`*) – class to convert from mutwo pitches to midi pitches. Default to *`MutwoPitchToMidiPitch`*.
- **`ticks_per_beat`** (*`int`*) – Sets the timing precision of the midi file. From the mido documentation: "Typical values range from 96 to 480 but some use even more ticks per beat".
- **`instrument_name`** (*`str`*) – Sets the midi instrument of all channels.
- **`tempo_envelope`** (`core_events.TempoEnvelope`) – All Midi files should specify their tempo. The default value of mutwo is 120 BPM (this is also the value that is assumed by any midi-file-reading-software if no tempo has been specified). Tempo changes are supported (and will be written to the resulting midi file).

Example:

```
>>> from mutwo.converters.frontends import midi
>>> from mutwo.ext.parameters import pitches
>>> # midi file converter that assign a middle c to all events
>>> midi_converter = midi.EventToMidiFile(
>>>     simple_event_to_pitch_list=lambda event: (pitches.WesternPitch('c'),)
>>> )
```

Disclaimer:

The current implementation doesn't support glissandi yet (only static pitches), time-signatures (the written time signature is always 4/4 for now) and dynamically changing tempo (ritardando or accelerando).

`convert`(*event_to_convert, path*)

Render a Midi file to the converters path attribute from the given event.

Parameters

- **`event_to_convert`** (*`Union[core_events.SimpleEvent, core_events.SequentialEvent[core_events.SimpleEvent], core_events.SimultaneousEvent[core_events.SequentialEvent[core_events.SimpleEvent]]]`*) – The given event that shall be translated to a Midi file.
- **`path`** (*`str`*) – where to write the midi file. The typical file type extension '.mid' is recommended, but not mandatory.

Return type

None

The following example generates a midi file that contains a simple ascending pentatonic scale:

```
>>> from mutwo.events import basic, music
>>> from mutwo.ext.parameters import pitches
```

(continues on next page)

(continued from previous page)

```
>>> from mutwo.converters.frontends import midi
>>> ascending_scale = basic.SequentialEvent(
>>>     [
>>>         music.NoteLike(pitches.WesternPitch(pitch), duration=1, volume=0.5)
>>>         for pitch in 'c d e g a'.split(' ')
>>>     ]
>>> )
>>> midi_converter = midi.EventToMidiFile(
>>>     available_midi_channel_tuple=(0,)
>>> )
>>> midi_converter.convert(ascending_scale, 'ascending_scale.mid')
```

Disclaimer: when passing nested structures, make sure that the nested object matches the expected type. Unlike other mutwo converter classes (like `mutwo.converters.core_converters.TempoConverter`) *EventToMidiFile* can't convert infinitely nested structures (due to the particular way how Midi files are defined). The deepest potential structure is a *mutwo.core_events.SimultaneousEvent* (representing the complete MidiFile) that contains *mutwo.core_events.SequentialEvent* (where each `SequentialEvent` represents one MidiTrack) that contains *mutwo.core_events.SimpleEvent* (where each `SimpleEvent` represents one midi note). If only one `SequentialEvent` is send, this `SequentialEvent` will be read as one MidiTrack in a MidiFile. If only one `SimpleEvent` get passed, this `SimpleEvent` will be interpreted as one MidiEvent (note_on and note_off) inside one MidiTrack inside one MidiFile.

mutwo.midi_converters.configurations

Configure the midi converters behaviour

`DEFAULT_AVAILABLE_MIDI_CHANNEL_TUPLE = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)`

default value for `available_midi_channel_tuple` in *MidiFileConverter*

`DEFAULT_CONTROL_MESSAGE_TUPLE_ATTRIBUTE_NAME = 'control_message_tuple'`

The expected attribute name of a *mutwo.core_events.SimpleEvent* for control messages.

`DEFAULT_MAXIMUM_PITCH_BEND_DEVIATION_IN_CENTS = 200`

default value for `maximum_pitch_bend_deviation_in_cents` in *MidiFileConverter*

`DEFAULT_MIDI_FILE_TYPE = 1`

default value for `midi_file_type` in *MidiFileConverter*

`DEFAULT_MIDI_INSTRUMENT_NAME = 'Acoustic Grand Piano'`

default value for `midi_instrument_name` in *MidiFileConverter*

`DEFAULT_N_MIDI_CHANNELS_PER_TRACK = 1`

default value for `n_midi_channels_per_track` in *MidiFileConverter*

`DEFAULT_TEMPO_ENVELOPE:` *TempoEnvelope* `= TempoEnvelope([SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 1), value = TempoPoint(BPM = 120, reference = 1)), SimpleEvent(curve_shape = 0, duration = DirectDuration(duration = 0), value = TempoPoint(BPM = 120, reference = 1))])`

default value for `tempo_envelope` in *MidiFileConverter*

`DEFAULT_TICKS_PER_BEAT = 480`

default value for `ticks_per_beat` in *MidiFileConverter*

mutwo.midi_converters.constants

Values that are defined by the midi file standard.

`MidiNote`

MidiNote type alias

`MidiPitch`

MidiPitch type alias

`MidiVelocity`

MidiVelocity type alias

`PitchBend`

PitchBend type alias

`ALLOWED_MIDI_CHANNEL_TUPLE = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)`

midi channels that are allowed (following the standard midi file definition).

`MAXIMUM_PITCH_BEND = 16382`

the highest allowed value for midi pitch bend

`MIDI_TEMPO_FACTOR = 1000000`

factor to multiply beats-in-seconds to get beats-in-microseconds (which is the tempo unit for midi)

`NEUTRAL_PITCH_BEND = 8191`

the value for midi pitch bend when the resulting pitch doesn't change

mutwo.midi_version

Table of content

`VERSION = '0.8.1'`

The version of the package `mutwo.midi`.

mutwo.music_converters

Table of content

Object	Documentation
mutwo.music_converters.GraceNotesConverter	Apply grace notes and after grace notes on `core_events.abc.Event`.
mutwo.music_converters.LoudnessToAmplitude	Make an approximation of the needed amplitude for a perceived Loudness.
mutwo.music_converters.RhythmicalStrataToIndispensability	Builds metrical indispensability for a rhythmical strata.
mutwo.music_converters.SimpleEventToPitchList	
mutwo.music_converters.SimpleEventToVolume	
mutwo.music_converters.SimpleEventToLyric	
mutwo.music_converters.SimpleEventToPlayingIndicatorCollection	
mutwo.music_converters.SimpleEventToNotationIndicatorCollection	
mutwo.music_converters.SimpleEventToGraceNoteSequentialEvent	
mutwo.music_converters.SimpleEventToAfterGraceNoteSequentialEvent	
mutwo.music_converters.MutwoParameterDictToPitchList	
mutwo.music_converters.MutwoParameterDictToVolume	
mutwo.music_converters.MutwoParameterDictToPlayingIndicatorCollection	
mutwo.music_converters.MutwoParameterDictToNotationIndicatorCollection	
mutwo.music_converters.MutwoParameterDictToGraceNoteSequentialEvent	
mutwo.music_converters.MutwoParameterDictToAfterGraceNoteSequentialEvent	
mutwo.music_converters.MutwoParameterDictToNoteLike	Convert a dict of mutwo parameters to a *mutwo.music_events.NoteLike*
mutwo.music_converters.ImproveWesternPitchListSequenceReadability	Adjust accidentals of pitches for a tonal-like visual representation
mutwo.music_converters.PlayingIndicatorConverter	Abstract base class to apply *PlayingIndicator* on a *SimpleEvent*.
mutwo.music_converters.ArpeggioConverter	Apply arpeggio on *SimpleEvent*.
mutwo.music_converters.StacattoConverter	Apply staccato on *SimpleEvent*.
mutwo.music_converters.ArticulationConverter	Apply articulation on *SimpleEvent*.
mutwo.music_converters.TrillConverter	Apply trill on *SimpleEvent*.
mutwo.music_converters.PlayingIndicatorsConverter	Apply `PlayingIndicator` on any *Event*.
mutwo.music_converters.TwoPitchesToCommonHarmonicTuple	Find the common harmonics between two pitches.

class GraceNotesConverter(*minima_grace_notes_duration_factor=0.12*, *maxima_grace_notes_duration_factor=0.25*, *minima_number_of_grace_notes=1*, *maxima_number_of_grace_notes=4*, *simple_event_to_grace_note_sequential_event=<mutwo.music_converters.parsers.SimpleEventToGraceNoteSequentialEvent object>*, *simple_event_to_after_grace_note_sequential_event=<mutwo.music_converters.parsers.SimpleEventToAfterGraceNoteSequentialEvent object>*)

Bases: *EventConverter*

Apply grace notes and after grace notes on `core_events.abc.Event`.

Parameters

- **minima_grace_notes_duration_factor** (*float*) – Minimal percentage how much of the initial duration of the *SimpleEvent* shall be moved to the grace notes / after grace notes. This value has to be smaller than 0.5 (so that the `SimpleEvent` have a duration > 0 if it has both: grace notes and after grace notes) and bigger than 0 (so that the grace notes or after grace notes have a duration > 0). Default to 0.12.
- **maxima_grace_notes_duration_factor** (*float*) – Maxima percentage how much of the initial duration of the *SimpleEvent* shall be moved to the grace notes / after grace notes. This value has to be smaller than 0.5 (so that the `SimpleEvent` have a duration > 0 if it has both: grace notes and after grace notes) and bigger than 0 (so that the grace notes or after grace notes have a duration > 0). Default to 0.25.
- **minima_number_of_grace_notes** (*int*) – For how many events in the grace note or after grace note container shall the *minima_grace_notes_duration_factor* be applied. Default to 1.
- **maxima_number_of_grace_notes** (*int*) – For how many events in the grace note or after grace note container shall the *maxima_number_of_grace_notes* be applied. Default to 4.

- **simple_event_to_grace_note_sequential_event** (*Callable[[core_events.SimpleEvent], core_events.SequentialEvent[core_events.SimpleEvent]]*) – Function which receives as an input a *SimpleEvent* and returns a *SequentialEvent*. By default the function will ask the event for a `grace_note_sequential_event` attribute, because by default *~mutwo.events.music.NoteLike* objects are expected.
- **simple_event_to_after_grace_note_sequential_event** (*Callable[[core_events.SimpleEvent], core_events.SequentialEvent[core_events.SimpleEvent]]*) – Function which receives as an input a *SimpleEvent* and returns a *SequentialEvent*. By default the function will ask the event for a `grace_note_sequential_event` attribute, because by default *~mutwo.events.music.NoteLike* objects are expected.

`convert`(*event_to_convert*)

Apply grace notes and after grace notes of all `SimpleEvent`.

Parameters

event_to_convert (core_events.abc.Event) – The event which grace notes and after grace notes shall be converted to normal events in the upper `SequentialEvent`.

Return type

Event

class LoudnessToAmplitude(*loudspeaker_frequency_response=Envelope([SimpleEvent(curve_shape=0, duration=DirectDuration(duration=2000), value=80), SimpleEvent(curve_shape=0, duration=DirectDuration(duration=0), value=80)]), interpolation_order=4*)

Bases: *Converter*

Make an approximation of the needed amplitude for a perceived Loudness.

Parameters

- **loudspeaker_frequency_response** (mutwo.core_events.Envelope) – Optionally the frequency response of the used loudspeaker can be added for balancing out uneven curves in the loudspeakers frequency response. The frequency response is defined with a `core_events.Envelope` object.
- **interpolation_order** (*int*) – The interpolation order of the equal loudness contour interpolation.

The converter works best with pure sine waves.

`convert`(*perceived_loudness_in_sone, frequency*)

Calculates the needed amplitude to reach a particular loudness for the entered frequency.

Parameters

- **perceived_loudness_in_sone** (*core_constants.Real*) – The subjectively perceived loudness that the resulting signal shall have (in the unit *Sone*).
- **frequency** (*Union[float, Fraction, int]*) – A frequency in Hertz for which the necessary amplitude shall be calculated.

Returns

Return the amplitude for a sine tone to reach the converters loudness when played with the entered frequency.

Return type

Union[float, *Fraction*, int]

Example:

```
>>> from mutwo.converters import symmetrical
>>> loudness_converter = symmetrical.loudness.LoudnessToAmplitudeConverter(1)
>>> loudness_converter.convert(200)
0.009364120303317933
>>> loudness_converter.convert(50)
0.15497924558613232
```

class RhythmicalStrataToIndispensability

Bases: *Converter*

Builds metrical indispensability for a rhythmical strata.

This technique has been described by Clarence Barlow in *On the Quantification of Harmony and Metre* (1992). The technique aims to model the weight of single beats in a particular metre. It allocates each beat of a metre to a specific value that describes the *indispensability* of a beat: the higher the assigned value, the more accented the beat.

convert(*rhythmical_strata_to_convert*)

Convert indispensability for each beat of a particular metre.

Parameters

rhythmical_strata_to_convert (*Sequence[int]*) – The rhythmical strata defines the metre for which the indispensability shall be calculated. The rhythmical strata is a list of prime numbers which product is the amount of available beats within the particular metre. Earlier prime numbers in the rhythmical strata are considered to be more important than later prime numbers.

Returns

A tuple of a integer for each beat of the respective metre where each integer describes how accented the particular beat is (the higher the number, the more important the beat).

Return type

tuple[int, …]

Example:

```
>>> from mutwo.converters import symmetrical
>>> metricity_converter = symmetrical.metricities.RhythmicalStrataToIndispensability()
>>> metricity_converter.convert((2, 3))  # time signature 3/4
(5, 0, 3, 1, 4, 2)
>>> metricity_converter.convert((3, 2))  # time signature 6/8
(5, 0, 2, 4, 1, 3)
```

class SimpleEventToPitchList(*attribute_name=None*, *exception_value=[]*)

Bases: *SimpleEventToAttribute*

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (*list[*mutwo.music_parameters.abc.Pitch*]*) –

class SimpleEventToVolume(*attribute_name=None*, *exception_value=DirectVolume(0)*)

Bases: *SimpleEventToAttribute*

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (Volume) –

class SimpleEventToLyric(*attribute_name=None*, *exception_value=<mutwo.music_parameters.lyrics.DirectLyric object>*)

Bases: *SimpleEventToAttribute*

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (Volume) –

class SimpleEventToPlayingIndicatorCollection(*attribute_name=None*, *exception_value=None*)

Bases: SimpleEventToAttributeWithDefaultValue

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (*Optional[*NotationIndicatorCollection*]*) –

class SimpleEventToNotationIndicatorCollection(*attribute_name=None*, *exception_value=None*)

Bases: SimpleEventToAttributeWithDefaultValue

Parameters

- **attribute_name** (*Optional[str]*) –
- **exception_value** (*Optional[*NotationIndicatorCollection*]*) –

class SimpleEventToGraceNoteSequentialEvent(*attribute_name=None*, *exception_value=SequentialEvent([])*)

Bases: *SimpleEventToAttribute*

Parameters

- **attribute_name** (*Optional[str]*) –

- `exception_value` (SequentialEvent) –

class `SimpleEventToAfterGraceNoteSequentialEvent`(*attribute_name=None, exception_value=SequentialEvent([])*)

Bases: *SimpleEventToAttribute*

Parameters

- `attribute_name` (*Optional[str]*) –
- `exception_value` (SequentialEvent) –

class `MutwoParameterDictToPitchList`(*pitch_list_to_search_name=None, pitch_list_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `pitch_list_to_search_name` (*Optional[str]*) –
- `pitch_list_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToVolume`(*volume_to_search_name=None, volume_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `volume_to_search_name` (*Optional[str]*) –
- `volume_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToPlayingIndicatorCollection`(*playing_indicator_collection_to_search_name=None, playing_indicator_collection_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `playing_indicator_collection_to_search_name` (*Optional[str]*) –
- `playing_indicator_collection_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToNotationIndicatorCollection`(*notation_indicator_collection_to_search_name=None, notation_indicator_collection_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `notation_indicator_collection_to_search_name` (*Optional[str]*) –
- `notation_indicator_collection_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToGraceNoteSequentialEvent`(*grace_note_sequential_event_to_search_name=None, grace_note_sequential_event_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `grace_note_sequential_event_to_search_name` (*Optional[str]*) –
- `grace_note_sequential_event_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToAfterGraceNoteSequentialEvent`(*after_grace_note_sequential_event_to_search_name=None, after_grace_note_sequential_event_keyword_name=None*)

Bases: *MutwoParameterDictToKeywordArgument*

Parameters

- `after_grace_note_sequential_event_to_search_name` (*Optional[str]*) –
- `after_grace_note_sequential_event_keyword_name` (*Optional[str]*) –

class `MutwoParameterDictToNoteLike`(*mutwo_parameter_dict_to_keyword_argument_sequence=None, simple_event_class=<class 'mutwo.music_events.music.NoteLike'>*)

Bases: *MutwoParameterDictToSimpleEvent*

Convert a dict of mutwo parameters to a *mutwo.music_events.NoteLike*

Parameters

- **mutwo_parameter_dict_to_keyword_argument_sequence** (*Optional[Sequence[MutwoParameterDictToKeywordArgu* – A sequence of `MutwoParameterDictToKeywordArgument`. Default to *None*.
- **simple_event_class** (*Type[core_events.SimpleEvent]*) – Default to *mutwo.music_events.NoteLike*.

class ImproveWesternPitchListSequenceReadability(*simultaneous_pitch_weight=1, sequential_pitch_weight=0.7, iteration_count=10000, optimizer_class=<class 'gradient_free_optimizers.optimizers.global_opt.random_search.RandomSearchOptimizer'>, verbosity_list=[], seed=100*)

Bases: *Converter*

Adjust accidentals of pitches for a tonal-like visual representation

Parameters

- **simultaneous_pitch_weight** (*float*) – Factor with which the weights of the resulting fitness from pitches of the same pitch list will be multiplied. Use higher value if a good form of simultaneous pitches is more important for you. Default to 1.
- **sequential_pitch_weight** (*float*) – Factor with which the weights of the resulting fitness from pitches of neighbouring pitch lists will be multiplied. Use higher value if a good form of sequential pitches is more important for you. Default to 0.7.
- **iteration_count** (*int*) – How many iterations the heuristic algorithm shall run. Use higher number for better (but slower) results. Default to 10000.
- **optimizer_class** (*BaseOptimizer*) – Sets optimizer class used within the converter. This can be any optimizer defined in the gradient_free_optimizers package. Default to `gradient_free_optimizers.RandomSearchOptimizer`.
- **verbosity_list** (*list[str]*) – From 'gradient_free_optimizers' documentation: "The verbosity list determines what part of the optimization information will be printed in the command line.". The complete list would be *["progress_bar", "print_results", "print_times"]*. Default to [] (no logging, silent).
- **seed** (*Optional[int]*) – The random seed used within the algorithm. Can be *None* for not-deterministic output. Default to 100.

Type

gradient_free_optimizers.optimizers.base_optimizer.BaseOptimizer,

This converter aims to adjust :class:`music_parameters.WesternPitch`s in order to improve the quality of western notation created with these pitches. Non-tonal music should be notated in a way to make it look as tonal as possible (e.g. it should notate intervals musicians are used to, it should avoid augmented or diminished intervals). The converter aims to maximize simple intervals (without changing the actual pitch content) by heuristic techniques. The converter may not return the best solution, but a very good approximation.

Disclaimer:

This converter doesn't work with microtonal pitches! This is due to the fact that *mutwo.music_parameters.WesternPitchInterval* doesn't support microtonal pitches yet.

PitchNameTupleToIntervalQualityDict

alias of `dict[tuple[str], bool]`

PitchVariantListTuple

alias of `tuple[list[tuple[`*WesternPitch*`, ...]], ...]`

RealSearchSpace

alias of `dict[str, tuple[`*WesternPitch*`]]`

SearchSpace

alias of `dict[str, int]`

convert(*western_pitch_list_sequence_to_convert*)

Simplify western pitch notation.

Parameters

western_pitch_list_sequence_to_convert (*Sequence[list[music_parameters.WesternPitch]]*) – A sequence filled with lists of *mutwo.music_parameters.WesternPitch*. The pitches will be simplified.

Returns

A tuple with lists that contain `music_parameters.WesternPitch`. The raw pitch content will be the same as the input data, but the accidentals and diatonic pitch class names may differ.

Return type

tuple[list[*mutwo.music_parameters.pitches.WesternPitch.WesternPitch*], ...]

class PlayingIndicatorConverter(*simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPlayingIndicatorCollection object>*)

Bases: *Converter*

Abstract base class to apply *PlayingIndicator* on a *SimpleEvent*.

Parameters

simple_event_to_playing_indicator_collection (*Callable[[*core_events.SimpleEvent*],* music_parameters.PlayingIndicatorCollection*], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.PlayingIndicatorCollection* object. By default it asks the Event for its `playing_indicator_collection` attribute (because by default `mutwo.ext.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their playing_indicator_collection property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.

To write a new PlayingIndicatorConverter the abstract method `_apply_playing_indicator()` and the abstract properties *playing_indicator_name* and *default_playing_indicator* have to be overridden.

convert(*simple_event_to_convert*)

Apply PlayingIndicator on simple_event.

Parameters

simple_event_to_convert (core_events.SimpleEvent) – The event which shall be converted.

Return type

SequentialEvent[SimpleEvent]

abstract property default_playing_indicator: *PlayingIndicator*

abstract property playing_indicator_name: str

class ArpeggioConverter(*duration_for_each_attack=0.1, simple_event_to_pitch_list=<mutwo.music_converters.parsers.SimpleEventToPitchList object>, simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPlayingIndicatorCollection object>, set_pitch_list_for_simple_event=<function ArpeggioConverter.<lambda>>*)

Bases: *PlayingIndicatorConverter*

Apply arpeggio on *SimpleEvent*.

Parameters

- **duration_for_each_attack** (*constants.DurationType*) – Set how long each attack of the Arpeggio lasts. Default to 0.1.
- **simple_event_to_pitch_list** (*Callable[[*core_events.SimpleEvent*],* music_parameters.abc.Pitch*], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a tuple that contains pitch objects (objects that inherit from *mutwo.music_parameters.abc.Pitch*). By default it asks the Event for its `pitch_list` attribute (because by default `mutwo.ext.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their pitch property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no pitch can be extracted), mutwo will assume an event without any pitches.
- **simple_event_to_playing_indicator_collection** (*Callable[[*core_events.SimpleEvent*],* music_parameters.PlayingIndicatorCollection*,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.PlayingIndicatorCollection* object. By default it asks the Event for its `playing_indicator_collection` attribute (because by default `mutwo.ext.events.music.NoteLike` objects are expected). When using different Event classes than `NoteLike` with a different name for their playing_indicator_collection property, this argument should be overridden. If the function call raises an `AttributeError` (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.
- **set_pitch_list_for_simple_event** (*Callable[[*core_events.SimpleEvent*, list[*music_parameters.abc.Pitch*]], None]*) – Function which assigns a list of *Pitch* objects to a *SimpleEvent*. By default the function assigns the passed pitches to the `pitch_list` attribute (because by default `mutwo.ext.events.music.NoteLike` objects are expected).

property default_playing_indicator: *PlayingIndicator*

property playing_indicator_name: str

class StacattoConverter(*factor=0.5, allowed_articulation_name_sequence=('staccato', '.'), simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPlayingIndicatorCollection object>*)

Bases: *PlayingIndicatorConverter*

Apply staccato on *SimpleEvent*.

Parameters

- **factor** (*float*) –
- **allowed_articulation_name_sequence** (*Sequence[str]*) –
- **simple_event_to_playing_indicator_collection** (*Callable[[core_events.SimpleEvent], music_parameters.PlayingIndicatorCollection,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.PlayingIndicatorCollection* object. By default it asks the Event for its playing_indicator_collection attribute (because by default mutwo.ext.events.music.NoteLike objects are expected). When using different Event classes than NoteLike with a different name for their playing_indicator_collection property, this argument should be overridden. If the function call raises an AttributeError (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.

property default_playing_indicator: *PlayingIndicator*

property playing_indicator_name: **str**

class ArticulationConverter(*articulation_name_tuple_to_playing_indicator_converter={('staccato', '.'): <mutwo.music_converters.playing_indicators.StacattoConverter object>}, simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPlayingIndicatorCollection object>*)

Bases: *PlayingIndicatorConverter*

Apply articulation on *SimpleEvent*.

Parameters

- **articulation_name_tuple_to_playing_indicator_converter** (*dict[tuple[str, ...], PlayingIndicatorConverter]*) –
- **simple_event_to_playing_indicator_collection** (*Callable[[core_events.SimpleEvent], music_parameters.PlayingIndicatorCollection,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a *mutwo.music_parameters.PlayingIndicatorCollection* object. By default it asks the Event for its playing_indicator_collection attribute (because by default mutwo.ext.events.music.NoteLike objects are expected). When using different Event classes than NoteLike with a different name for their playing_indicator_collection property, this argument should be overridden. If the function call raises an AttributeError (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.

property default_playing_indicator: *PlayingIndicator*

property playing_indicator_name: **str**

class TrillConverter(*trill_size=Fraction(1, 16), simple_event_to_pitch_list=<mutwo.music_converters.parsers.SimpleEventToPitchList object>, simple_event_to_playing_indicator_collection=<mutwo.music_converters.parsers.SimpleEventToPitchList object>*)

Bases: *PlayingIndicatorConverter*

Apply trill on *SimpleEvent*.

Parameters

- **trill_size** (*constants.DurationType*) –
- **simple_event_to_pitch_list** (*Callable[[core_events.SimpleEvent], music_parameters.abc.Pitch], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a tuple that contains pitch objects (objects that inherit from *mutwo.music_parameters.abc.Pitch*). By default it asks the Event for its pitch_list attribute (because by default mutwo.ext.events.music.NoteLike objects are expected). When using different Event classes than NoteLike with a different name for their pitch property, this argument should be overridden. If the function call raises an AttributeError (e.g. if no pitch can be extracted), mutwo will assume an event without any pitches.
- **simple_event_to_playing_indicator_collection** (*Callable[[core_events.SimpleEvent], music_parameters.PlayingIndicatorCollection,], optional*) – Function to extract from a *mutwo.core_events.SimpleEvent* a mutwo.ext.parameters.playing_indicators.PlayingIndicatorCollection object. By default it asks the Event for its playing_indicator_collection attribute (because by default mutwo.ext.events.music.NoteLike objects are expected). When using different Event classes than NoteLike with a different name for their playing_indicator_collection property, this argument should be overridden. If the function call raises an

`AttributeError` (e.g. if no playing indicator collection can be extracted), mutwo will build a playing indicator collection from *DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS*.

property default_playing_indicator: *PlayingIndicator*

property playing_indicator_name: str

class PlayingIndicatorsConverter(*playing_indicator_converter_sequence*)

Bases: *SymmetricalEventConverter*

Apply `PlayingIndicator` on any *Event*.

Parameters

playing_indicator_converter_sequence (*Sequence[*PlayingIndicatorConverter*]*) – A sequence of *PlayingIndicatorConverter* which shall be applied on each *SimpleEvent*.

convert(*event_to_convert*)

Parameters

event_to_convert (Event) –

Return type

Event

class TwoPitchesToCommonHarmonicTuple(*tonality*, *lowest_partial*, *highest_partial*)

Bases: *Converter*

Find the common harmonics between two pitches.

Parameters

- **tonality** (*Optional[bool]*) – `True` for finding common harmonics, `False` for finding common subharmonics and `None` for finding common pitches between the harmonics of the first pitch and the subharmonics of the second pitch.
- **lowest_partial** (*int*) – The lowest partial to get investigated. Shouldn't be smaller than 1.
- **highest_partial** (*int*) – The highest partial to get investigated. Shouldn't be bigger than 1.

convert(*pitch_pair_to_examine*)

Parameters

pitch_pair_to_examine (*tuple[*mutwo.music_parameters.pitches.JustIntonationPitch.JustIntonationPitch, mutwo.music_parameters.pitches.JustIntonationPitch.JustIntonationPitch*]*) –

Return type

tuple[*mutwo.music_parameters.pitches.CommonHarmonic.CommonHarmonic*, ...]

mutwo.music_converters.configurations

Configure the default behaviour of *mutwo.music_converters*

DEFAULT_AFTER_GRACE_NOTE_SEQUENTIAL_EVENT_KEYWORD_NAME = 'after_grace_note_sequential_event'

Default value for **:param:`after_grace_note_sequential_event_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToAfterGraceNoteSequentialEvent`

DEFAULT_AFTER_GRACE_NOTE_SEQUENTIAL_EVENT_TO_SEARCH_NAME = 'after_grace_note_sequential_event'

Default value for **:param:`after_grace_note_sequential_event_to_search_name`** parameter in *mutwo.music_converters.MutwoParameterDictToAfterGraceNoteSequentialEvent* and default value for **:param:`attribute_name`** in *mutwo.music_converters.SimpleEventToAfterGraceNoteSequentialEvent*.

DEFAULT_GRACE_NOTE_SEQUENTIAL_EVENT_KEYWORD_NAME = 'grace_note_sequential_event'

Default value for **:param:`grace_note_sequential_event_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToGraceNoteSequentialEvent`

DEFAULT_GRACE_NOTE_SEQUENTIAL_EVENT_TO_SEARCH_NAME = 'grace_note_sequential_event'

Default value for **:param:`grace_note_sequential_event_to_search_name`** parameter in *mutwo.music_converters.MutwoParameterDictToGraceNoteSequentialEvent* and default value for **:param:`attribute_name`** in *mutwo.music_converters.SimpleEventToGraceNoteSequentialEvent*.

`DEFAULT_LYRIC_TO_SEARCH_NAME = 'lyric'`

Default value for **:param:`lyric_to_search_name`** parameter in `mutwo.music_converters.MutwoParameterDictToLyric` and default value for **:param:`attribute_name`** in *`mutwo.music_converters.SimpleEventToLyric`*.

`DEFAULT_NOTATION_INDICATOR_COLLECTION_KEYWORD_NAME = 'notation_indicator_collection'`

Default value for **:param:`notation_indicator_collection_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToNotationIndicatorCollection`

`DEFAULT_NOTATION_INDICATOR_COLLECTION_TO_SEARCH_NAME = 'notation_indicator_collection'`

Default value for **:param:`notation_indicator_collection_to_search_name`** parameter in *`mutwo.music_converters.MutwoParameterDictToNotationIndicatorCollection`* and default value for **:param:`attribute_name`** in *`mutwo.music_converters.SimpleEventToNotationIndicatorCollection`*.

`DEFAULT_PITCH_LIST_KEYWORD_NAME = 'pitch_list'`

Default value for **:param:`pitch_list_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToPitchList`

`DEFAULT_PITCH_LIST_TO_SEARCH_NAME = 'pitch_list'`

Default value for **:param:`pitch_list_to_search_name`** parameter in *`mutwo.music_converters.MutwoParameterDictToPitchList`* and default value for **:param:`attribute_name`** in *`mutwo.music_converters.SimpleEventToPitchList`*.

`DEFAULT_PLAYING_INDICATOR_COLLECTION_KEYWORD_NAME = 'playing_indicator_collection'`

Default value for **:param:`playing_indicator_collection_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToPlayingIndicatorCollection`

`DEFAULT_PLAYING_INDICATOR_COLLECTION_TO_SEARCH_NAME = 'playing_indicator_collection'`

Default value for **:param:`playing_indicator_collection_to_search_name`** parameter in *`mutwo.music_converters.MutwoParameterDictToPlayingIndicatorCollection`* and default value for **:param:`attribute_name`** in *`mutwo.music_converters.SimpleEventToPlayingIndicatorCollection`*.

`DEFAULT_VOLUME_KEYWORD_NAME = 'volume'`

Default value for **:param:`volume_keyword_name`** parameter in `mutwo.core_converters.MutwoParameterDictToVolume`

`DEFAULT_VOLUME_TO_SEARCH_NAME = 'volume'`

Default value for **:param:`volume_to_search_name`** parameter in *`mutwo.music_converters.MutwoParameterDictToVolume`* and default value for **:param:`attribute_name`** in *`mutwo.music_converters.SimpleEventToVolume`*.

mutwo.music_converters.constants

Several constants which are used for the loudness converter module.

`AUDITORY_THRESHOLD_AT_1KHZ = 2e-05`

Roughly the sound of a mosquito flying 3 m away (see https://en.wikipedia.org/wiki/Sound_pressure).

mutwo.music_events

Table of content

Object	Documentation
`mutwo.music_events.NoteLike`	NoteLike represents traditional discreet musical objects.

`class NoteLike`(*pitch_list='c', duration=1, volume='mf', grace_note_sequential_event=None, after_grace_note_sequential_event=None, playing_indicator_collection=None, notation_indicator_collection=None, lyric=<mutwo.music_parameters.lyrics.DirectLyric object>*)

Bases: *`SimpleEvent`*

NoteLike represents traditional discreet musical objects.

Parameters

- **pitch_list** (*Optional[Union[Pitch, Sequence, float, Fraction, int]]*) – The pitch or pitches of the event. This can be a pitch object (any class that inherits from `mutwo.music_parameters.abc.Pitch`) or a list of pitch objects. Furthermore mutwo supports syntactic sugar to convert other objects on the fly to pitch objects: Atring can be read as pitch class names to build *mutwo.music_parameters.WesternPitch* objects or as ratios to build *mutwo.music_parameters.JustIntonationPitch* objects. Fraction will also build *mutwo.music_parameters.JustIntonationPitch* objects. Other numbers (integer and float) will be read as pitch class numbers to make *mutwo.music_parameters.WesternPitch* objects.
- **duration** (*Union[float, Fraction, int]*) – The duration of `NoteLike`. This can be any number. The unit of the duration is up to the interpretation of the user and the respective converter routine that will be used.
- **volume** (*Union[Volume, float, Fraction, int, str]*) – The volume of the event. Can either be a object of *mutwo.music_parameters.abc.Volume*, a number or a string. If the number ranges from 0 to 1, mutwo automatically generates a *mutwo.music_parameters.DirectVolume* object (and the number will be interpreted as the amplitude). If the number is smaller than 0, automatically generates a *mutwo.music_parameters.volumes.DecibelVolume* object (and the number will be interpreted as decibel). If the argument is a string, *mutwo* will try to initialise a *mutwo.music_parameters.volumes.WesternVolume* object.
- **grace_note_sequential_event** (*core_events.SequentialEvent[NoteLike]*) –
- **after_grace_note_sequential_event** (*core_events.SequentialEvent[NoteLike]*) –
- **playing_indicator_collection** (*music_parameters.playing_indicator_collection.PlayingIndicatorCollection*) – A `PlayingIndicatorCollection`. Playing indicators alter the sound of *NoteLike* (e.g. tremolo, fermata, pizzicato).
- **notation_indicator_collection** (*music_parameters.notation_indicator_collection.NotationIndicatorCollection*) – A `NotationIndicatorCollection`. Notation indicators alter the visual representation of *NoteLike* (e.g. ottava, clefs) without affecting the resulting sound.
- **lyric** (*core_parameters.abc.Lyric*) –

By default mutwo doesn't differentiate between Tones, Chords and Rests, but rather simply implements one general class which can represent any of the mentioned definitions (e.g. a NoteLike object with several pitches may be called a 'Chord' and a NoteLike object with only one pitch may be called a 'Tone').

Example:

```
>>> from mutwo import music_parameters
>>> from mutwo import music_events
>>> tone = music_events.NoteLike(music_parameters.WesternPitch('a'), 1, 1)
>>> other_tone = music_events.NoteLike('3/2', 1, 0.5)
>>> chord = music_events.NoteLike(
    [music_parameters.WesternPitch('a'), music_parameters.JustIntonationPitch('3/2')], 1, 1
)
>>> other_chord = music_events.NoteLike('c4 dqs3 10/7', 1, 3)
```

property after_grace_note_sequential_event: *SequentialEvent*[*SimpleEvent*]

`core_events.SequentialEvent` after *NoteLike*

property grace_note_sequential_event: *SequentialEvent*[*SimpleEvent*]

`core_events.SequentialEvent` before *NoteLike*

property pitch_list: Any

The pitch or pitches of the event.

property volume: Any

The volume of the event.

mutwo.music_events.configurations

Set default values for *mutwo.music_events.NoteLike*.

`DEFAULT_NOTATION_INDICATORS_COLLECTION_CLASS`
: Default value for `notation_indicator_collection` in *NoteLike*

`DEFAULT_PLAYING_INDICATORS_COLLECTION_CLASS`
: Default value for `playing_indicator_collection` in *NoteLike*

mutwo.music_generators

Table of content

Object	Documentation
mutwo.music_generators.make_product_pitch	Make `JustIntonationPitch` from the product of one, two or more number_sequence.
mutwo.music_generators.make_common_product_set_scale	Make common product set scale as described in Wilsons letter to Fokker.
mutwo.music_generators. make_wilsons_brun_euclidean_algorithm_generator	Make constant structure scale with Wilsons adaption of Bruns euclidean algorithm.

`make_product_pitch`(*number_sequence*, *tonality*, *normalize=False*)

Make `JustIntonationPitch` from the product of one, two or more number_sequence.

Parameters

- `number_sequence` (*Sequence[int]*) – The number which shall be multiplied to make a new pitch.
- `tonality` (*bool*) – `True` for putting the resulting product to the numerator of the frequency ratio and `False` for putting the resulting product to the denominator.
- `normalize` (*bool, optional*) – `True` to normalize the new pitch to the middle octave. Default to `False`.

Return type

JustIntonationPitch

`make_common_product_set_scale`(*number_sequence*, *n_combinations*, *tonality*, *normalize=False*)

Make common product set scale as described in Wilsons letter to Fokker.

Parameters

- `number_sequence` (*Sequence[int]*) – The number_sequence which will be combined to single music_parameters.
- `n_combinations` (*int*) – How many number_sequence will be combined for each pitch.
- `tonality` (*bool*) – `True` for otonality and `False` for utonality.
- `normalize` (*bool*) – `True` if music_parameters.shall become normalized to the same octave.

Return type

tuple[*mutwo.music_parameters.pitches.JustIntonationPitch.JustIntonationPitch*, ...]

Example:

```
>>> from mutwo.generators import wilson
>>> wilson.make_common_product_set_scale((3, 5, 7, 9), 2, True)
(JustIntonationPitch(15),
 JustIntonationPitch(21),
 JustIntonationPitch(27),
 JustIntonationPitch(35),
 JustIntonationPitch(45),
 JustIntonationPitch(63))
>>> wilson.make_common_product_set_scale((3, 5, 7, 9), 2, False)
```

(continues on next page)

(continued from previous page)

```
(JustIntonationPitch(1/15),
 JustIntonationPitch(1/21),
 JustIntonationPitch(1/27),
 JustIntonationPitch(1/35),
 JustIntonationPitch(1/45),
 JustIntonationPitch(1/63))
```

make_wilsons_brun_euclidean_algorithm_generator(*pitch_tuple*, *subtraction_index=1*, *direction_forward=True*, *direction_reverse=False*)

Make constant structure scale with Wilsons adaption of Bruns euclidean algorithm.

Parameters

- **pitch_tuple** (*tuple[*music_parameters.JustIntonationPitch, music_parameters.JustIntonationPitch, music_parameters.JustIntonationPitch*],)*) – The initial seed composed of three individual music_parameters. The biggest pitch will be the period of the repeating scale, therefore it is recommended to use `music_parameters.JustIntonationPitch("2/1")` here (if one desires an octave repeating scale).
- **subtraction_index** (*int*) – Set to 1 if the largest interval should be subtracted by the second interval. Set to 2 if the largest interval should be subtracted by the smallest interval.
- **direction_forward** (*bool*) – Set to `True` if the algorithm should include the normal sorted replacement of an interval. Default to `True`.
- **direction_reverse** (*bool*) – Set to `True` if the algorithm should include the reversed replacement of an interval. Default to `False`.

Returns

Generator which returns a list of intervals. Accumulate the intervals from `music_parameters.JustIntonationPitch("1/1")` to get the scale music_parameters.

Return type

Generator

Example:

```
>>> from mutwo.ext.parameters import pitches
>>> from mutwo.ext.generators import wilson
>>> wilsons_brun_euclidean_algorithm_generator = (
>>>     wilson.make_wilsons_brun_euclidean_algorithm_generator(
>>>         (
>>>             music_parameters.JustIntonationPitch("2/1"),
>>>             music_parameters.JustIntonationPitch("3/2"),
>>>             music_parameters.JustIntonationPitch("5/4"),
>>>         )
>>>     )
>>> )
>>> next(wilsons_brun_euclidean_algorithm_generator)
((JustIntonationPitch(2),),)
>>> next(wilsons_brun_euclidean_algorithm_generator)
((JustIntonationPitch(3/2), JustIntonationPitch(4/3)),)
>>> next(wilsons_brun_euclidean_algorithm_generator)
((JustIntonationPitch(4/3), JustIntonationPitch(9/8), JustIntonationPitch(4/3)),)
```

```
TUNEABLE_INTERVAL_TO_DIFFICULTY_DICT = {(): 0, (-3, 0, 0, 0, 0, 0, 0, 0, 1): 1, (-3, 0, 0, 0, 0, 0, 0, 1): 2,
(-3, 0, 0, 0, 0, 1): 1, (-3, 0, 0, 0, 1): 2, (-3, 0, 2): 2, (-3, 1, 1): 2, (-3, 3): 2, (-2, -1, 0, 0, 0, 0, 0,
0, 1): 2, (-2, 0, 0, 0, 0, 0, 0, 0, 1): 1, (-2, 0, 0, 0, 0, 0, 0, 1): 1, (-2, 0, 0, 0, 0, 0, 1): 1, (-2, 0, 0,
0, 0, 1): 0, (-2, 0, 0, 0, 1): 0, (-2, 0, 0, 1): 0, (-2, 0, 1): 0, (-2, 0, 2): 1, (-2, 1, 0, 1): 1, (-2, 1, 1):
0, (-2, 2): 0, (-2, 3): 2, (-1, -1, 0, 0, 0, 0, 0, 0, 1): 2, (-1, -1, 0, 0, 0, 0, 0, 1): 2, (-1, -1, 0, 0, 0, 0,
1): 1, (-1, -1, 0, 0, 0, 1): 1, (-1, -1, 0, 0, 1): 1, (-1, -1, 0, 1): 0, (-1, -1, 2): 2, (-1, 0, -1, 0, 0, 1):
1, (-1, 0, 0, 0, 0, 1): 0, (-1, 0, 0, 0, 1): 0, (-1, 0, 0, 1): 0, (-1, 0, 1): 0, (-1, 1): 0, (-1, 1, 1): 0, (-1,
2): 0, (0, -2, 0, 0, 0, 1): 2, (0, -2, 0, 0, 1): 2, (0, -1, 0, 0, 0, 0, 0, 0, 1): 0, (0, -1, 0, 0, 0, 0, 0, 1):
0, (0, -1, 0, 0, 0, 0, 1): 0, (0, -1, 0, 0, 0, 1): 0, (0, -1, 0, 0, 1): 0, (0, -1, 0, 1): 0, (0, -1, 1): 0, (0,
0, -1, 0, 0, 0, 0, 0, 1): 1, (0, 0, -1, 0, 0, 0, 0, 1): 1, (0, 0, -1, 0, 0, 0, 1): 1, (0, 0, -1, 0, 0, 1): 1,
(0, 0, -1, 0, 1): 1, (0, 0, -1, 1): 0, (0, 0, 0, -1, 0, 0, 1): 1, (0, 0, 0, -1, 0, 1): 1, (0, 0, 0, -1, 1): 1,
(0, 0, 0, 1): 0, (0, 0, 1): 0, (0, 1): 0, (0, 1, -1, 1): 2, (0, 2, -1): 0, (0, 2, 0, -1): 0, (0, 3, 0, -1): 2,
(1,): 0, (1, -2, 0, 1): 2, (1, -1, 0, 0, 1): 0, (1, -1, 0, 1): 0, (1, -1, 1): 0, (1, 0, -1, 0, 0, 1): 2, (1, 0,
-1, 0, 1): 2, (1, 0, -1, 1): 0, (1, 0, 0, -1, 1): 2, (1, 0, 1, -1): 1, (1, 1): 0, (1, 1, -1): 0, (1, 2, -1): 0,
(1, 2, 0, -1): 1, (2,): 0, (2, -2, 0, 1): 2, (2, -1): 0, (2, -1, 1): 0, (2, 0, -1, 1): 1, (2, 0, 1, -1): 1, (2,
1, -1): 0, (2, 1, 0, -1): 1, (3,): 0, (3, -1): 0, (3, 0, -1): 0, (3, 0, 0, -1): 1, (3, 1, -1): 1, (3, 1, 0, -1):
2, (4, -1): 0, (4, 0, -1): 0, (4, 0, 0, 0, -1): 2}
```

Tuneable Just Intonation Intervals sorted by difficulty, according to Marc Sabat.

```
TUNEABLE_INTERVAL_TUPLE = (JustIntonationPitch('1/1'), JustIntonationPitch('8/7'), JustIntonationPitch('7/6'),
JustIntonationPitch('6/5'), JustIntonationPitch('11/9'), JustIntonationPitch('5/4'), JustIntonationPitch('9/7'),
JustIntonationPitch('13/10'), JustIntonationPitch('4/3'), JustIntonationPitch('11/8'),
JustIntonationPitch('7/5'), JustIntonationPitch('10/7'), JustIntonationPitch('13/9'),
JustIntonationPitch('16/11'), JustIntonationPitch('3/2'), JustIntonationPitch('14/9'),
JustIntonationPitch('11/7'), JustIntonationPitch('8/5'), JustIntonationPitch('13/8'),
JustIntonationPitch('5/3'), JustIntonationPitch('12/7'), JustIntonationPitch('7/4'), JustIntonationPitch('9/5'),
JustIntonationPitch('11/6'), JustIntonationPitch('13/7'), JustIntonationPitch('15/8'),
JustIntonationPitch('23/12'), JustIntonationPitch('2/1'), JustIntonationPitch('13/6'),
JustIntonationPitch('11/5'), JustIntonationPitch('9/4'), JustIntonationPitch('7/3'),
JustIntonationPitch('19/8'), JustIntonationPitch('12/5'), JustIntonationPitch('17/7'),
JustIntonationPitch('5/2'), JustIntonationPitch('18/7'), JustIntonationPitch('13/5'),
JustIntonationPitch('8/3'), JustIntonationPitch('11/4'), JustIntonationPitch('14/5'),
JustIntonationPitch('17/6'), JustIntonationPitch('20/7'), JustIntonationPitch('23/8'),
JustIntonationPitch('3/1'), JustIntonationPitch('28/9'), JustIntonationPitch('25/8'),
JustIntonationPitch('22/7'), JustIntonationPitch('19/6'), JustIntonationPitch('16/5'),
JustIntonationPitch('13/4'), JustIntonationPitch('10/3'), JustIntonationPitch('27/8'),
JustIntonationPitch('17/5'), JustIntonationPitch('24/7'), JustIntonationPitch('7/2'),
JustIntonationPitch('18/5'), JustIntonationPitch('11/3'), JustIntonationPitch('15/4'),
JustIntonationPitch('19/5'), JustIntonationPitch('23/6'), JustIntonationPitch('27/7'),
JustIntonationPitch('4/1'), JustIntonationPitch('25/6'), JustIntonationPitch('21/5'),
JustIntonationPitch('17/4'), JustIntonationPitch('13/3'), JustIntonationPitch('22/5'),
JustIntonationPitch('9/2'), JustIntonationPitch('23/5'), JustIntonationPitch('14/3'),
JustIntonationPitch('19/4'), JustIntonationPitch('24/5'), JustIntonationPitch('5/1'),
JustIntonationPitch('26/5'), JustIntonationPitch('21/4'), JustIntonationPitch('16/3'),
JustIntonationPitch('11/2'), JustIntonationPitch('28/5'), JustIntonationPitch('17/3'),
JustIntonationPitch('23/4'), JustIntonationPitch('6/1'), JustIntonationPitch('25/4'),
JustIntonationPitch('19/3'), JustIntonationPitch('13/2'), JustIntonationPitch('20/3'),
JustIntonationPitch('27/4'), JustIntonationPitch('7/1'), JustIntonationPitch('22/3'),
JustIntonationPitch('15/2'), JustIntonationPitch('23/3'), JustIntonationPitch('8/1'))
```

Tuneable Just Intonation Intervals according to Marc Sabat.

mutwo.music_parameters

Table of content

Object	Documentation
`mutwo.music_parameters.OctaveAmbitus`	
`mutwo.music_parameters.Comma`	A tuning comma.
`mutwo.music_parameters.CommaCompound`	Collection of tuning commas.
`mutwo.music_parameters.DirectLyric`	Lyric which is directly initialised by its phonetic representation
`mutwo.music_parameters.LanguageBasedLyric`	Lyric based on a natural language.
`mutwo.music_parameters.LanguageBasedSyllable`	Syllable based on a natural language.
`mutwo.music_parameters.DirectPitchInterval`	Simple interval class which gets directly assigned by its cents value
`mutwo.music_parameters.WesternPitchInterval`	Model intervals by using European music theory based representations
`mutwo.music_parameters.DirectPitch`	A simple pitch class that gets directly initialised by its frequency.
`mutwo.music_parameters.JustIntonationPitch`	Pitch that is defined by a frequency ratio and a reference pitch.
`mutwo.music_parameters.Partial`	Abstract representation of a harmonic spectrum partial.
`mutwo.music_parameters.EqualDividedOctavePitch`	Pitch that is tuned to an Equal divided octave tuning system.
`mutwo.music_parameters.WesternPitch`	Pitch with a traditional Western nomenclature.
`mutwo.music_parameters.MidiPitch`	Pitch that is defined by its midi pitch number.
`mutwo.music_parameters.CommonHarmonic`	`JustIntonationPitch` which is the common harmonic between two or more other pitches.
`mutwo.music_parameters.DirectVolume`	A simple volume class that gets directly initialised by its amplitude.
`mutwo.music_parameters.DecibelVolume`	A simple volume class that gets directly initialised by decibel.
`mutwo.music_parameters.WesternVolume`	Volume with a traditional Western nomenclature.
`mutwo.music_parameters.BarLine`	BarLine(abbreviation: Optional[str] = None)
`mutwo.music_parameters.Clef`	Clef(name: Optional[str] = None)
`mutwo.music_parameters.Ottava`	Ottava(n_octaves: Optional[int] = 0)
`mutwo.music_parameters.MarginMarkup`	MarginMarkup(content: Optional[str] = None, context: Optional[str] = 'Staff')
`mutwo.music_parameters.Markup`	Markup(content: Optional[str] = None, direction: Optional[str] = None)
`mutwo.music_parameters.RehearsalMark`	RehearsalMark(markup: Optional[str] = None)
`mutwo.music_parameters.NotationIndicatorCollection`	NotationIndicatorCollection(bar_line: mutwo.music_parameters.notation_indicators.BarLine = <factory>, clef: mutwo.music_parameters.notation_indicators.Clef = <factory>, ottava: mutwo.music_parameters.notation_indicators.Ottava = <factory>, margin_markup: mutwo.music_parameters.notation_indicators.MarginMarkup = <factory>, markup: mutwo.music_parameters.notation_indicators.Markup = <factory>, rehearsal_mark: mutwo.music_parameters.notation_indicators.RehearsalMark = <factory>)
`mutwo.music_parameters.Tremolo`	Tremolo(n_flags: Optional[int] = None)
`mutwo.music_parameters.Articulation`	Articulation(name: Optional[Literal['accent', 'marcato', 'staccatissimo', 'espressivo', 'staccato', 'tenuto', 'portato', 'upbow', 'downbow', 'flageolet', 'thumb', 'lheel', 'rheel', 'ltoe', 'rtoe', 'open', 'halfopen', 'snappizzicato', 'stopped', 'turn', 'reverseturn', 'trill', 'prall', 'mordent', 'prallprall', 'prallmordent', 'upprall', 'downprall', 'upmordent', 'downmordent', 'pralldown', 'prallup', 'lineprall', 'signumcongruentiae', 'shortfermata', 'fermata', 'longfermata', 'verylongfermata', 'segno', 'coda', 'varcoda', '^', '+', '-', '|', '>', '.', '_']] = None)
`mutwo.music_parameters.Arpeggio`	Arpeggio(direction: Optional[Literal['up', 'down']] = None)

continues on next page

Table 3 – continued from previous page

Object	Documentation
mutwo.music_parameters.Pedal	Pedal(pedal_type: Optional[Literal['sustain', 'sostenuto', 'corda']] = None, pedal_activity: Optional[bool] = True)
mutwo.music_parameters.StringContactPoint	StringContactPoint(contact_point: Optional[Literal['dietro ponticello', 'molto sul ponticello', 'molto sul tasto', 'ordinario', 'pizzicato', 'ponticello', 'sul ponticello', 'sul tasto', 'col legno tratto', 'd.p.', 'm.s.p', 'm.s.t.', 'ord.', 'pizz.', 'p.', 's.p.', 's.t.', 'c.l.t.']] = None)
mutwo.music_parameters.Ornamentation	Ornamentation(direction: Optional[Literal['up', 'down']] = None, n_times: int = 1)
mutwo.music_parameters.BendAfter	BendAfter(bend_amount: Optional[float] = None, minimum_length: Optional[float] = 3, thickness: Optional[float] = 3)
mutwo.music_parameters.ArtificalHarmonic	ArtificalHarmonic(n_semitones: Optional[int] = None)
mutwo.music_parameters.PreciseNaturalHarmonic	PreciseNaturalHarmonic(string_pitch: Optional[mutwo.music_parameters.pitches.WesternPitch.WesternPitch] = None, played_pitch: Optional[mutwo.music_parameters.pitches.WesternPitch.WesternPitch] = None, harmonic_note_head_style: bool = True, parenthesize_lower_note_head: bool = False)
mutwo.music_parameters.Fermata	Fermata(fermata_type: Optional[Literal['shortfermata', 'fermata', 'longfermata', 'verylongfermata']] = None)
mutwo.music_parameters.Hairpin	Hairpin(symbol: Optional[Literal['<', '>', '<>', '!']] = None, niente: bool = False)
mutwo.music_parameters.Trill	Trill(pitch: Optional[mutwo.music_parameters.abc.Pitch] = None)
mutwo.music_parameters.WoodwindFingering	WoodwindFingering(cc: Optional[Tuple[str, ...]] = None, left_hand: Optional[Tuple[str, ...]] = None, right_hand: Optional[Tuple[str, ...]] = None, instrument: str = 'clarinet')
mutwo.music_parameters.Cue	Cue for electronics etc.
mutwo.music_parameters.PlayingIndicatorCollection	PlayingIndicatorCollection(articulation: mutwo.music_parameters.playing_indicators.Articulation = <factory>, artifical_harmonic: mutwo.music_parameters.playing_indicators.ArtificalHarmonic = <factory>, arpeggio: mutwo.music_parameters.playing_indicators.Arpeggio = <factory>, bartok_pizzicato: mutwo.music_parameters.abc.PlayingIndicator = <factory>, bend_after: mutwo.music_parameters.playing_indicators.BendAfter = <factory>, breath_mark: mutwo.music_parameters.abc.PlayingIndicator = <factory>, cue: mutwo.music_parameters.playing_indicators.Cue = <factory>, duration_line_dashed: mutwo.music_parameters.abc.PlayingIndicator = <factory>, duration_line_triller: mutwo.music_parameters.abc.PlayingIndicator = <factory>, fermata: mutwo.music_parameters.playing_indicators.Fermata = <factory>, glissando: mutwo.music_parameters.abc.PlayingIndicator = <factory>, hairpin: mutwo.music_parameters.playing_indicators.Hairpin = <factory>, natural_harmonic: mutwo.music_parameters.abc.PlayingIndicator = <factory>, laissez_vibrer: mutwo.music_parameters.abc.PlayingIndicator = <factory>, ornamentation: mutwo.music_parameters.playing_indicators.Ornamentation = <factory>, pedal: mutwo.music_parameters.playing_indicators.Pedal = <factory>, prall: mutwo.music_parameters.abc.PlayingIndicator = <factory>, precise_natural_harmonic: mutwo.music_parameters.playing_indicators.PreciseNaturalHarmonic = <factory>, string_contact_point: mutwo.music_parameters.playing_indicators.StringContactPoint = <factory>, tie: mutwo.music_parameters.abc.PlayingIndicator = <factory>, tremolo: mutwo.music_parameters.playing_indicators.Tremolo = <factory>, trill: mutwo.music_parameters.playing_indicators.Trill = <factory>, woodwind_fingering: mutwo.music_parameters.playing_indicators.WoodwindFingering = <factory>)

class OctaveAmbitus(*minima_pitch*, *maxima_pitch*)

Bases: *PitchAmbitus*

Parameters

- **minima_pitch** (Pitch) –
- **maxima_pitch** (Pitch) –

pitch_to_period(*pitch*)

Parameters
pitch (Pitch) –

Return type
PitchInterval

`class` **Comma**(*ratio*)

Bases: `object`

A tuning comma.

Parameters
ratio (*Fraction*) –

`property` **ratio: Fraction**

`class` **CommaCompound**(*prime_to_exponent_dict*, *prime_to_comma_dict*)

Bases: `Iterable`[*Comma*]

Collection of tuning commas.

Parameters

- **prime_to_exponent_dict** (*dict[int, int]*) –
- **prime_to_comma_dict** (*Optional[dict[int,* mutwo.music_parameters.commas.Comma*]]*) –

`property` **prime_to_exponent_dict: dict[int, int]**

`property` **ratio: Fraction**

`class` **DirectLyric**(*phonetic_representation*)

Bases: *Lyric*

Lyric which is directly initialised by its phonetic representation

Parameters
phonetic_representation (*str*) – The phonetic representation of the text.

In this class the *written_representation* is simply equal to *phonetic_representation*.

`property` **phonetic_representation: str**

`property` **written_representation: str**

Get text as it would be written in natural language

`class` **LanguageBasedLyric**(*written_representation*, *language_code=None*)

Bases: *Lyric*

Lyric based on a natural language.

Parameters

- **written_representation** (*str*) – The text.
- **language_code** (*Optional[str]*) – The code for the language of the text. If this is *None* the constant *mutwo.music_parameters.configurations.DEFAULT_LANGUAGE_CODE* will be used. Default to *None*.

`property` **language_code: str**

`property` **phonetic_representation: str**

`property` **written_representation: str**

Get text as it would be written in natural language

class LanguageBasedSyllable(*is_last_syllable*, **args*, ***kwargs*)

Bases: *Syllable*, *LanguageBasedLyric*

Syllable based on a natural language.

Parameters

- **is_last_syllable** (*bool*) – *True* if it is the last syllable of a word and *False* if it isn't the last syllable
- **written_representation** (*str*) – The text.
- **language_code** (*Optional[str]*) – The code for the language of the text. If this is *None* the constant *mutwo.music_parameters.configurations.DEFAULT_LANGUAGE_CODE* will be used. Default to *None*.

Warning:

It is a known bug that a split word (syllables) and the word itself will return different values for `phonetic_representation`. For instance:

```
>>> LanguageBasedLyric('hello').phonetic_representation
"h@l@U"
>>> # And now splitted to syllables:
>>> LanguageBasedSyllable('hel').phonetic_representation
"he5"
>>> LanguageBasedSyllable('lo').phonetic_representation
"l@U"
```

class DirectPitchInterval(*interval*)

Bases: *PitchInterval*

Simple interval class which gets directly assigned by its cents value

Parameters

interval (*float*) – Defines how big or small the interval is (in cents).

Example:

```
>>> from mutwo import music_parameters
>>> rising_octave = music_parameters.DirectPitchInterval(1200)
>>> falling_minor_third = music_parameters.DirectPitchInterval(-300)
```

property interval: float

class WesternPitchInterval(*interval_name_or_semitone_count='p1'*)

Bases: *PitchInterval*

Model intervals by using European music theory based representations

Parameters

interval_name_or_semitone_count (*Union[str, core_constants.Real]*) – Can be either an interval name (a string) or a number for semitones. When using an interval name is should have the form: QUALITY-IS_FALLING-TYPE, e.g. for having a rising perfect fourth (where 'fourth' is the type and 'perfect' the quality) you can write "p4". For a falling perfect fourth it would be "p-4". The interval names are equal to the specification used in the python library music21. Please also consult the specification of the quality abbreviations at `mutwo.music_parameters.configurations.WESTERN_PITCH_INTERVAL_QUALITY_NAME_TO_ABBREVIATION_DICT` and the specification of the *is-interval-falling* indicator `mutwo.music_parameters.configurations.FALLING_WESTERN_PITCH_INTERVAL_INDICATOR`. Both can be changed by the user. Default to 'p1'.

This class is particularly useful in combination with *mutwo.music_parameters.WesternPitch*.

Disclaimer:

Although *mutwo.music_parameters.WesternPitch* does support microtones, *WesternPitchInterval* does not.

Example:

```
>>> from mutwo import music_parameters
>>> perfect_fifth = music_parameters.WesternPitchInterval('p5')
>>> falling_major_third = music_parameters.WesternPitchInterval('M-3')
>>> minor_third = music_parameters.WesternPitchInterval('m3')
>>> falling_octave = music_parameters.WesternPitchInterval(-12)
>>> augmented_octave = music_parameters.WesternPitchInterval('A8')
>>> very_diminished_sixth = music_parameters.WesternPitchInterval('dddd6')
```

`inverse()`

Return type
WesternPitchInterval

`inverse_direction(`*mutate=False*`)`

Makes falling interval to rising and vice versa.

Example:

```
>>> from mutwo import music_parameters
>>> music_parameters.WesternPitchInterval('m3').inverse_direction()
WesternPitchInterval('m-3')
```

Parameters
`mutate` (*bool*) –

Return type
WesternPitchInterval

`static is_interval_type_imperfect(`*interval_type*`)`

Parameters
`interval_type` (*str*) –

Return type
bool

`static is_interval_type_perfect(`*interval_type*`)`

Parameters
`interval_type` (*str*) –

Return type
bool

`property can_be_simplified: bool`

True if interval could be written in a simpler way, *False* otherwise.

`property diatonic_pitch_class_count: int`

How many diatonic pitch classes have to be moved

`property interval: float`

`property interval_quality: str`

The abbreviation of its quality (e.g. augmented, perfect, …).

`property interval_quality_cent_deviation: float`

Get cent deviation defined by the interval quality.

`property interval_quality_tuple: tuple[str, ...]`

Parsed the interval_quality abbreviation to their full names.

`property interval_type: str`

The base interval type (e.g. octave, prime, second, …).

`property interval_type_base_type: str`

`property interval_type_cent_deviation: float`

Get cent deviation defined by the interval type.

`property is_imperfect_interval: bool`

Return *True* if interval is imperfect and otherwise *False*.

With 'imperfect' all intervals are included which can have the interval qualities 'augmented', 'diminished', 'minor' and 'major'.

This excludes intervals as prime, fourth, … which have the 'perfect' quality.

`property is_interval_rising: bool`

Return *True* if the interval is upwards and *False* if it falls

property is_perfect_interval: bool

Return *True* if interval is perfect and otherwise *False*.

With 'perfect' all intervals are included which can have the interval qualities 'augmented', 'diminished' and 'perfect'.

This excludes intervals as sixth, thirds, … which have 'minor' and 'major' qualities.

property name: str

Full interval name

property semitone_count: float

class DirectPitch(*frequency*, **args*, ***kwargs*)

Bases: *Pitch*

A simple pitch class that gets directly initialised by its frequency.

Parameters

frequency (*core_constants.Real*) – The frequency of the `DirectPitch` object.

May be used when a converter class needs a pitch object, but there is no need or desire for a complex abstraction of the respective pitch (that classes like `JustIntonationPitch` or `WesternPitch` offer).

Example:

```
>>> from mutwo.music_parameters import pitches
>>> my_pitch = pitches.DirectPitch(440)
```

add(*pitch_interval*, *mutate=False*)

Parameters

- **pitch_interval** (PitchInterval) –
- **mutate** (*bool*) –

Return type

DirectPitch

property frequency: float

The frequency of the pitch.

class JustIntonationPitch(*ratio_or_exponent_tuple='1/1'*, *concert_pitch=None*, **args*, ***kwargs*)

Bases: *Pitch*, *PitchInterval*

Pitch that is defined by a frequency ratio and a reference pitch.

Parameters

- **ratio_or_exponent_tuple** (*Union[str, fractions.Fraction, Iterable[int]]*) – The frequency ratio of the `JustIntonationPitch`. This can either be a string that indicates the frequency ratio (for instance: "1/1", "3/2", "9/2", etc.), or a `fractions.Fraction` object that indicates the frequency ratio (for instance: `fractions.Fraction(3, 2)`, `fractions.Fraction(7, 4)`) or an Iterable that is filled with integer that represents the exponent_tuple of the respective prime numbers of the decomposed frequency ratio. The prime numbers are rising and start with 2. Therefore the tuple `(2, 0, -1)` would return the frequency ratio `4/5` because `(2 ** 2) * (3 ** 0) * (5 ** -1) = 4/5`.
- **concert_pitch** (*ConcertPitch*) – The reference pitch of the tuning system (the pitch for a frequency ratio of 1/1). Can either be another `Pitch` object or any number to indicate a particular frequency in Hertz.

The resulting frequency is calculated by multiplying the frequency ratio with the respective reference pitch.

Example:

```
>>> from mutwo.music_parameters import pitches
>>> # 3 different variations of initialising the same pitch
>>> pitches.JustIntonationPitch('3/2')
>>> import fractions
>>> pitches.JustIntonationPitch(fractions.Fraction(3, 2))
>>> pitches.JustIntonationPitch((-1, 1))
>>> # using a different concert pitch
>>> pitches.JustIntonationPitch('7/5', concert_pitch=432)
```

add(*pitch_interval*)

Add *JustIntonationPitch* to current pitch.

Parameters

- **other** – The *JustIntonationPitch* to add to the current pitch.
- **pitch_interval** (PitchInterval) –

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p = pitches.JustIntonationPitch('3/2')
>>> p.add(pitches.JustIntonationPitch('3/2'))
>>> p
JustIntonationPitch(9/4)
```

get_closest_pythagorean_pitch_name(*reference='a'*)

Parameters

reference (*str*) –

Return type

str

get_pitch_interval(*pitch_to_compare*)

Get `PitchInterval` between itself and other pitch

Parameters

pitch_to_compare (Pitch) – The pitch which shall be compared to the active pitch.

Returns

`PitchInterval` between

Return type

PitchInterval

Example:

```
>>> from mutwo import music_parameters
>>> a4 = music_parameters.DirectPitch(frequency=440)
>>> a5 = music_parameters.DirectPitch(frequency=880)
>>> a4.get_pitch_interval(a5)
DirectPitchInterval(cents = 1200)
```

intersection(*other*, *strict=False*)

Make intersection with other *JustIntonationPitch*.

Parameters

- **other** (JustIntonationPitch) – The *JustIntonationPitch* to build the intersection with.
- **strict** (*bool*) – If set to `True` only exponent_tuple are included into the intersection if their value is equal. If set to `False` the method will also include exponent_tuple if both pitches own them on the same axis but with different values (the method will take the smaller exponent).

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p0 = pitches.JustIntonationPitch('5/3')
>>> p0.intersection(pitches.JustIntonationPitch('7/6'))
>>> p0
JustIntonationPitch(1/3)
>>> p1 = pitches.JustIntonationPitch('9/7')
>>> p1.intersection(pitches.JustIntonationPitch('3/2'))
```

(continues on next page)

(continued from previous page)

```
>>> p1
JustIntonationPitch(3/1)
>>> p2 = pitches.JustIntonationPitch('9/7')
>>> p2.intersection(pitches.JustIntonationPitch('3/2'), strict=True)
>>> p2
JustIntonationPitch(1/1)
```

inverse(*axis=None*)

Inverse current pitch on given axis.

Parameters

axis (*JustIntonationPitch, optional*) – The *JustIntonationPitch* from which the pitch shall be inversed.

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p = pitches.JustIntonationPitch('3/2')
>>> p.inverse()
>>> p
JustIntonationPitch(2/3)
```

move_to_closest_register(*reference*)

Parameters

reference (JustIntonationPitch) –

Return type

JustIntonationPitch

normalize(*prime=2*)

Normalize *JustIntonationPitch*.

Parameters

prime (*int*) – The normalization period (2 for octave, 3 for twelfth, ...). Default to 2.

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p = pitches.JustIntonationPitch('12/2')
>>> p.normalize()
>>> p
JustIntonationPitch(3/2)
```

register(*octave*)

Move *JustIntonationPitch* to the given octave.

Parameters

octave (*int*) – 0 for the octave from 1/1 to 2/1, negative values for octaves below 1/1 and positive values for octaves above 2/1.

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p = pitches.JustIntonationPitch('3/2')
>>> p.register(1)
>>> p
JustIntonationPitch(6/2)
>>> p.register(-1)
>>> p
JustIntonationPitch(3/4)
```

(continues on next page)

(continued from previous page)

```
>>> p.register(0)
>>> p
JustIntonationPitch(3/2)
```

subtract(*pitch_interval*)

Subtract *JustIntonationPitch* from current pitch.

Parameters

- **other** – The *JustIntonationPitch* to subtract from the current pitch.
- **pitch_interval** (PitchInterval) –

Return type

JustIntonationPitch

Example:

```
>>> from mutwo.music_parameters import pitches
>>> p = pitches.JustIntonationPitch('9/4')
>>> p.subtract(pitches.JustIntonationPitch('3/2'))
>>> p
JustIntonationPitch(3/2)
```

property blueprint: tuple[tuple[int, ...], ...]

property cent_deviation_from_closest_western_pitch_class: float

property closest_pythagorean_interval: *JustIntonationPitch*

property concert_pitch: *Pitch*

property denominator: int

Return the denominator of *JustIntonationPitch*.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1,))
>>> just_intonation_pitch0.denominator
1
```

property exponent_tuple: tuple

property factorised: tuple

Return factorised / decomposed version of itsef.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch0.factorised
(2, 2, 5)
>>> just_intonation_pitch1 = JustIntonationPitch("7/6")
>>> just_intonation_pitch1.factorised
(2, 3, 7)
```

property factorised_numerator_and_denominator: tuple

property frequency: float

property harmonic: int

Return the nth - harmonic / subharmonic the pitch may represent.

Returns

May be positive for harmonic and negative for subharmonic pitches. If the return - value is 0, the interval may occur neither between the first harmonic and any other pitch of the harmonic scale nor between the first subharmonic in the and any other pitch of the subharmonic scale.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1))
>>> just_intonation_pitch0.ratio
fractions.Fraction(3, 2)
>>> just_intonation_pitch0.harmonic
3
>>> just_intonation_pitch1 = JustIntonationPitch((-1,), 2)
>>> just_intonation_pitch1.harmonic
-3
```

property harmonicity_barlow: float

Calculate the barlow-harmonicity of an interval.

This implementation follows Clarence Barlows definition, given in 'The Ratio Book' (1992).

A higher number means a more harmonic interval / a less complex harmony.

barlow(1/1) is definied as infinite.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1,))
>>> just_intonation_pitch1 = JustIntonationPitch()
>>> just_intonation_pitch2 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch3 = JustIntonationPitch((0, 0, -1,))
>>> just_intonation_pitch0.harmonicity_barlow
0.27272727272727276
>>> just_intonation_pitch1.harmonicity_barlow # 1/1 is infinite harmonic
inf
>>> just_intonation_pitch2.harmonicity_barlow
0.11904761904761904
>>> just_intonation_pitch3.harmonicity_barlow
-0.10638297872340426
```

property harmonicity_euler: int

Return the 'gradus suavitatis' of euler.

A higher number means a less consonant interval / a more complicated harmony. euler(1/1) is definied as 1.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1,))
>>> just_intonation_pitch1 = JustIntonationPitch()
>>> just_intonation_pitch2 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch3 = JustIntonationPitch((0, 0, -1,))
>>> just_intonation_pitch0.harmonicity_euler
4
>>> just_intonation_pitch1.harmonicity_euler
1
>>> just_intonation_pitch2.harmonicity_euler
7
>>> just_intonation_pitch3.harmonicity_euler
8
```

property harmonicity_simplified_barlow: float

Calculate a simplified barlow-harmonicity of an interval.

This implementation follows Clarence Barlows definition, given in 'The Ratio Book' (1992), with the difference that only positive numbers are returned and that (1/1) is defined as 1 instead of infinite.

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1,))
>>> just_intonation_pitch1 = JustIntonationPitch()
>>> just_intonation_pitch2 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch3 = JustIntonationPitch((0, 0, -1,))
>>> just_intonation_pitch0.harmonicity_simplified_barlow
0.27272727272727276
>>> just_intonation_pitch1.harmonicity_simplified_barlow # 1/1 is not infinite but 1
```

(continues on next page)

(continued from previous page)

```
1
>>> just_intonation_pitch2.harmonicity_simplified_barlow
0.11904761904761904
>>> just_intonation_pitch3.harmonicity_simplified_barlow # positive return value
0.10638297872340426
```

property harmonicity_tenney: float

Calculate Tenneys harmonic distance of an interval

A higher number means a more consonant interval / a less complicated harmony.

tenney(1/1) is definied as 0.

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1,))
>>> just_intonation_pitch1 = JustIntonationPitch()
>>> just_intonation_pitch2 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch3 = JustIntonationPitch((0, 0, -1,))
>>> just_intonation_pitch0.harmonicity_tenney
2.584962500721156
>>> just_intonation_pitch1.harmonicity_tenney
0.0
>>> just_intonation_pitch2.harmonicity_tenney
4.321928094887363
>>> just_intonation_pitch3.harmonicity_tenney
-0.10638297872340426
```

property harmonicity_vogel: int

property harmonicity_wilson: int

property helmholtz_ellis_just_intonation_notation_commas: *CommaCompound*

Commas of JustIntonationPitch.

property interval: float

property numerator: int

Return the numerator of a JustIntonationPitch - object.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, -1,))
>>> just_intonation_pitch0.numerator
1
```

property occupied_primes: tuple

Return all occurring prime numbers of a JustIntonationPitch object.

property octave: int

property prime_tuple: tuple

Return ascending list of primes, until the highest contained Prime.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 1, 2))
>>> just_intonation_pitch0.exponent_tuple
(2, 3, 5)
>>> just_intonation_pitch1 = JustIntonationPitch((0, -1, 0, 0, 1), 1)
>>> just_intonation_pitch1.exponent_tuple
(2, 3, 5, 7, 11)
```

property primes_for_numerator_and_denominator: tuple

property ratio: Fraction

Return the JustIntonationPitch transformed to a Ratio.

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((0, 0, 1,))
>>> just_intonation_pitch0.ratio
fractions.Fraction(5, 4)
>>> just_intonation_pitch0 = JustIntonationPitch("3/2")
>>> just_intonation_pitch0.ratio
fractions.Fraction(3, 2)
```

property tonality: bool

Return the tonality (bool) of a JustIntonationPitch - object.

The tonality of a JustIntonationPitch - may be True (otonality) if the exponent of the highest occurring prime number is a positive number and False if the exponent is a negative number (utonality).

Example:

```
>>> just_intonation_pitch0 = JustIntonationPitch((-2. 1))
>>> just_intonation_pitch0.tonality
True
>>> just_intonation_pitch1 = JustIntonationPitch((-2, -1))
>>> just_intonation_pitch1.tonality
False
>>> just_intonation_pitch2 = JustIntonationPitch([])
>>> just_intonation_pitch2.tonality
True
```

class Partial(*nth_partial*, *tonality*)

Bases: `object`

Abstract representation of a harmonic spectrum partial.

Parameters

- **nth_partial** (*int*) – The number of the partial (starting with 1 for the root note).
- **tonality** (*bool*) – `True` for overtone and `False` for a (theoretical) undertone. Default to `True`.

Example:

```
>>> from mutwo.music_parameters import pitches
>>> strong_clarinet_partials = (
    pitches.Partial(1),
    pitches.Partial(3),
    pitches.Partial(5),
    pitches.Partial(7),
)
```

nth_partial: int

tonality: bool

class EqualDividedOctavePitch(*n_pitch_classes_per_octave*, *pitch_class*, *octave*, *concert_pitch_pitch_class*, *concert_pitch_octave*, *concert_pitch=None*, **args*, ***kwargs*)

Bases: *Pitch*

Pitch that is tuned to an Equal divided octave tuning system.

Parameters

- **n_pitch_classes_per_octave** (*int*) – how many pitch classes in each octave occur (for instance 12 for a chromatic system, 24 for quartertones, etc.)
- **pitch_class** (*core_constants.Real*) – The pitch class of the new *EqualDividedOctavePitch* object.
- **octave** (*int*) – The octave of the new *EqualDividedOctavePitch* object (where 0 is the middle octave, 1 is one octave higher and -1 is one octave lower).
- **concert_pitch_pitch_class** (*core_constants.Real*) – The pitch class of the reference pitch (for instance 9 in a chromatic 12 tone system where *a* should be the reference pitch).
- **concert_pitch_octave** (*int*) – The octave of the reference pitch.

- **concert_pitch** (*ConcertPitch*) – The frequency of the reference pitch (for instance 440 for a).

```
>>> from mutwo.music_parameters import pitches
>>> # making a middle `a`
>>> pitches.EqualDividedOctavePitch(12, 9, 4, 9, 4, 440)
```

add(*pitch_interval*)

Transposes the `EqualDividedOctavePitch` by n_pitch_classes_difference.

Parameters

pitch_interval (*Union[PitchInterval, float, Fraction, int]*) –

Return type

EqualDividedOctavePitch

subtract(*pitch_interval*)

Transposes the `EqualDividedOctavePitch` by n_pitch_classes_difference.

Parameters

pitch_interval (*Union[PitchInterval, float, Fraction, int]*) –

Return type

EqualDividedOctavePitch

property concert_pitch: ***Pitch***

The referential concert pitch for the respective pitch object.

property concert_pitch_pitch_class: Union[float, Fraction, int]

The pitch class of the referential concert pitch.

property frequency: float

property n_cents_per_step: float

This property describes how many cents are between two adjacent pitches.

property n_pitch_classes_per_octave: int

Defines in how many different pitch classes one octave get divided.

property pitch_class: Union[float, Fraction, int]

The pitch class of the pitch.

property step_factor

The factor with which to multiply a frequency to reach the next pitch.

class WesternPitch(*pitch_class_or_pitch_class_name=0, octave=4, concert_pitch_pitch_class=None, concert_pitch_octave=None, concert_pitch=None, *args, **kwargs*)

Bases: *EqualDividedOctavePitch*

Pitch with a traditional Western nomenclature.

Parameters

- **pitch_class_or_pitch_class_name** (*PitchClassOrPitchClassName*) – Name or number of the pitch class of the new `WesternPitch` object. The nomenclature is English (c, d, e, f, g, a, b). It uses an equal divided octave system in 12 chromatic steps. Accidentals are indicated by (s = sharp) and (f = flat). Further microtonal accidentals are supported (see `mutwo.music_parameters.constants.ACCIDENTAL_NAME_TO_PITCH_CLASS_MODIFICATION_DICT` for all supported accidentals).
- **octave** (*int*) – The octave of the new *WesternPitch* object. Indications for the specific octave follow the MIDI Standard where 4 is defined as one line.
- **concert_pitch_pitch_class** (*core_constants.Real*) –
- **concert_pitch_octave** (*int*) –
- **concert_pitch** (*ConcertPitch*) –

Example:

```
>>> from mutwo.music_parameters import pitches
>>> pitches.WesternPitch('cs', 4)  # c-sharp 4
>>> pitches.WesternPitch('aqs', 2)  # a-quarter-sharp 2
```

`add(pitch_interval)`

Transposes the `EqualDividedOctavePitch` by n_pitch_classes_difference.

Parameters

`pitch_interval` (*Union[str,* PitchInterval*, float, Fraction, int]*) –

Return type

WesternPitch

`classmethod from_midi_pitch_number(midi_pitch_number)`

Parameters

`midi_pitch_number` (*float*) –

Return type

WesternPitch

`get_pitch_interval(pitch_to_compare)`

Get `PitchInterval` between itself and other pitch

Parameters

`pitch_to_compare` (Pitch) – The pitch which shall be compared to the active pitch.

Returns

`PitchInterval` between

Return type

PitchInterval

Example:

```
>>> from mutwo import music_parameters
>>> a4 = music_parameters.DirectPitch(frequency=440)
>>> a5 = music_parameters.DirectPitch(frequency=880)
>>> a4.get_pitch_interval(a5)
DirectPitchInterval(cents = 1200)
```

`subtract(pitch_interval)`

Transposes the `EqualDividedOctavePitch` by n_pitch_classes_difference.

Parameters

`pitch_interval` (*Union[str,* PitchInterval*, float, Fraction, int]*) –

Return type

WesternPitch

`property accidental_name: str`

Only get accidental part of pitch name

`property diatonic_pitch_class_name: str`

Only get the diatonic part of the pitch name

`property enharmonic_pitch_tuple: tuple[mutwo.music_parameters.pitches.WesternPitch.WesternPitch, ...]`

Return pitches with equal frequency but different name.

Disclaimer:

This doesn't work in some corner cases yet (e.g. it won't find "css" for "eff")

`property is_microtonal: bool`

Return *True* if accidental isn't on chromatic grid.

`property name: str`

The name of the pitch in Western nomenclature.

`property pitch_class: Union[float, Fraction, int]`

The pitch class of the pitch.

`property pitch_class_name: str`

The name of the pitch class in Western nomenclature.

Mutwo uses the English nomenclature for pitch class names:

(c, d, e, f, g, a, b)

class MidiPitch(*midi_pitch_number*, **args*, ***kwargs*)

Bases: *Pitch*

Pitch that is defined by its midi pitch number.

Parameters

midi_pitch_number (*float*) – The midi pitch number of the pitch. Floating point numbers are possible for microtonal deviations from the chromatic scale.

Example:

```
>>> from mutwo.music_parameters import pitches
>>> middle_c = pitches.MidiPitch(60)
>>> middle_c_quarter_tone_high = pitches.MidiPitch(60.5)
```

add(*pitch_interval*, *mutate=False*)

Parameters

- **pitch_interval** (PitchInterval) –
- **mutate** (*bool*) –

Return type

MidiPitch

property frequency: float

property midi_pitch_number: float

The midi pitch number (from 0 to 127) of the pitch.

class CommonHarmonic(*partial_tuple*, *ratio_or_exponent_tuple='1/1'*, *concert_pitch=None*, **args*, ***kwargs*)

Bases: *JustIntonationPitch*

JustIntonationPitch which is the common harmonic between two or more other pitches.

Parameters

- **partials** (*tuple[*Partial*, ...]*) – Tuple which contains partial numbers.
- **ratio_or_exponent_tuple** (*Union[str, fractions.Fraction, Iterable[int]]*) – see the documentation of *JustIntonationPitch*
- **concert_pitch** (*Union[core_constants.Real,* music_parameters.abc.Pitch*]*) – see the documentation of *JustIntonationPitch*
- **partial_tuple** (*tuple[*Partial*, ...]*) –

class DirectVolume(*amplitude*)

Bases: *Volume*

A simple volume class that gets directly initialised by its amplitude.

Parameters

amplitude (*Union[float, Fraction, int]*) – The amplitude of the *DirectVolume* object.

May be used when a converter class needs a volume object, but there is no need or desire for a complex abstraction of the respective volume.

property amplitude: Union[float, Fraction, int]

class DecibelVolume(*decibel*)

Bases: *Volume*

A simple volume class that gets directly initialised by decibel.

Parameters

decibel (*Union[float, Fraction, int]*) – The decibel of the *DecibelVolume* object (should be from -120 to 0).

May be used when a converter class needs a volume object, but there is no need or desire for a complex abstraction of the respective volume.

property amplitude: Union[float, Fraction, int]

property decibel: Union[float, Fraction, int]

The decibel of the volume (from -120 to 0)

class WesternVolume(*name*, *minimum_decibel=None*, *maximum_decibel=None*)

Bases: `Volume`

Volume with a traditional Western nomenclature.

Parameters

- **name** (*str*) – Dynamic indicator in traditional Western nomenclature ('f', 'pp', 'mf', 'sfz', etc.). For a list of all supported indicators, see `mutwo.music_parameters.constants.DYNAMIC_INDICATOR_TUPLE`.
- **minimum_decibel** (*core_constants.Real, optional*) – The decibel value which is equal to the lowest dynamic indicator (ppppp).
- **maximum_decibel** (*core_constants.Real, optional*) – The decibel value which is equal to the highest dynamic indicator (fffff).

Example:

```
>>> from mutwo.music_parameters import volumes
>>> volumes.WesternVolume('fff')
WesternVolume(fff)
```

classmethod from_amplitude(*amplitude*)

Initialise *WesternVolume* from amplitude ratio.

Parameters

amplitude (*Union[float, Fraction, int]*) – The amplitude which shall be converted to a *WesternVolume* object.

Return type

WesternVolume

```
>>> from mutwo.music_parameters import volumes
>>> volumes.WesternVolume.from_amplitude(0.05)
WesternVolume(mp)
```

classmethod from_decibel(*decibel*)

Initialise *WesternVolume* from decibel.

Parameters

decibel (*Union[float, Fraction, int]*) – The decibel which shall be converted to a *WesternVolume* object.

Return type

WesternVolume

```
>>> from mutwo.music_parameters import volumes
>>> volumes.WesternVolume.from_decibel(-24)
WesternVolume(mf)
```

property amplitude: Union[float, Fraction, int]

property decibel: Union[float, Fraction, int]

The decibel of the volume (from -120 to 0)

property name: str

The western nomenclature name for dynamic.

For a list of all supported indicators, see `mutwo.music_parameters.constants.DYNAMIC_INDICATOR_TUPLE`.

class BarLine(*abbreviation: Optional[str] = None*)

Bases: `NotationIndicator`

Parameters

abbreviation (*Optional[str]*) –

abbreviation: Optional[str] = None

class Clef(*name: Optional[str] = None*)

Bases: `NotationIndicator`

Parameters

name (*Optional[str]*) –

`name: Optional[str] = None`

`class Ottava`(*n_octaves: Optional[int] = 0*)

Bases: `NotationIndicator`

Parameters

`n_octaves` (`Optional[int]`) –

`n_octaves: Optional[int] = 0`

`class MarginMarkup`(*content: Optional[str] = None, context: Optional[str] = 'Staff'*)

Bases: `NotationIndicator`

Parameters

- `content` (`Optional[str]`) –
- `context` (`Optional[str]`) –

`content: Optional[str] = None`

`context: Optional[str] = 'Staff'`

`class Markup`(*content: Optional[str] = None, direction: Optional[str] = None*)

Bases: `NotationIndicator`

Parameters

- `content` (`Optional[str]`) –
- `direction` (`Optional[str]`) –

`content: Optional[str] = None`

`direction: Optional[str] = None`

`class RehearsalMark`(*markup: Optional[str] = None*)

Bases: `NotationIndicator`

Parameters

`markup` (`Optional[str]`) –

`markup: Optional[str] = None`

`class NotationIndicatorCollection`(*bar_line: mutwo.music_parameters.notation_indicators.BarLine = <factory>, clef: mutwo.music_parameters.notation_indicators.Clef = <factory>, ottava: mutwo.music_parameters.notation_indicators.Ottava = <factory>, margin_markup: mutwo.music_parameters.notation_indicators.MarginMarkup = <factory>, markup: mutwo.music_parameters.notation_indicators.Markup = <factory>, rehearsal_mark: mutwo.music_parameters.notation_indicators.RehearsalMark = <factory>*)

Bases: `IndicatorCollection[NotationIndicator]`

Parameters

- `bar_line` (`BarLine`) –
- `clef` (`Clef`) –
- `ottava` (`Ottava`) –
- `margin_markup` (`MarginMarkup`) –
- `markup` (`Markup`) –
- `rehearsal_mark` (`RehearsalMark`) –

`bar_line: BarLine`

`clef: Clef`

`margin_markup: MarginMarkup`

`markup: Markup`

`ottava: Ottava`

`rehearsal_mark: RehearsalMark`

class `Tremolo`(*n_flags: Optional[int] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`n_flags` (*Optional[int]*) –

`n_flags: Optional[int] = None`

class `Articulation`(*name: Optional[Literal['accent', 'marcato', 'staccatissimo', 'espressivo', 'staccato', 'tenuto', 'portato', 'upbow', 'downbow', 'flageolet', 'thumb', 'lheel', 'rheel', 'ltoe', 'rtoe', 'open', 'halfopen', 'snappizzicato', 'stopped', 'turn', 'reverseturn', 'trill', 'prall', 'mordent', 'prallprall', 'prallmordent', 'upprall', 'downprall', 'upmordent', 'downmordent', 'pralldown', 'prallup', 'lineprall', 'signumcongruentiae', 'shortfermata', 'fermata', 'longfermata', 'verylongfermata', 'segno', 'coda', 'varcoda', '^', '+', '-', '|', '>', '.', '_']] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`name` (*Optional[Literal['accent', 'marcato', 'staccatissimo', 'espressivo', 'staccato', 'tenuto', 'portato', 'upbow', 'downbow', 'flageolet', 'thumb', 'lheel', 'rheel', 'ltoe', 'rtoe', 'open', 'halfopen', 'snappizzicato', 'stopped', 'turn', 'reverseturn', 'trill', 'prall', 'mordent', 'prallprall', 'prallmordent', 'upprall', 'downprall', 'upmordent', 'downmordent', 'pralldown', 'prallup', 'lineprall', 'signumcongruentiae', 'shortfermata', 'fermata', 'longfermata', 'verylongfermata', 'segno', 'coda', 'varcoda', '^', '+', '-', '|', '>', '.', '_']]*) –

`name: Optional[Literal['accent', 'marcato', 'staccatissimo', 'espressivo', 'staccato', 'tenuto', 'portato', 'upbow', 'downbow', 'flageolet', 'thumb', 'lheel', 'rheel', 'ltoe', 'rtoe', 'open', 'halfopen', 'snappizzicato', 'stopped', 'turn', 'reverseturn', 'trill', 'prall', 'mordent', 'prallprall', 'prallmordent', 'upprall', 'downprall', 'upmordent', 'downmordent', 'pralldown', 'prallup', 'lineprall', 'signumcongruentiae', 'shortfermata', 'fermata', 'longfermata', 'verylongfermata', 'segno', 'coda', 'varcoda', '^', '+', '-', '|', '>', '.', '_']] = None`

class `Arpeggio`(*direction: Optional[Literal['up', 'down']] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`direction` (*Optional[Literal['up', 'down']]*) –

`direction: Optional[Literal['up', 'down']] = None`

class `Pedal`(*pedal_type: Optional[Literal['sustain', 'sostenuto', 'corda']] = None, pedal_activity: Optional[bool] = True*)

Bases: *ImplicitPlayingIndicator*

Parameters

- `pedal_type` (*Optional[Literal['sustain', 'sostenuto', 'corda']]*) –
- `pedal_activity` (*Optional[bool]*) –

`pedal_activity: Optional[bool] = True`

`pedal_type: Optional[Literal['sustain', 'sostenuto', 'corda']] = None`

class `StringContactPoint`(*contact_point: Optional[Literal['dietro ponticello', 'molto sul ponticello', 'molto sul tasto', 'ordinario', 'pizzicato', 'ponticello', 'sul ponticello', 'sul tasto', 'col legno tratto', 'd.p.', 'm.s.p', 'm.s.t.', 'ord.', 'pizz.', 'p.', 's.p.', 's.t.', 'c.l.t.']] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`contact_point` (*Optional[Literal['dietro ponticello', 'molto sul ponticello', 'molto sul tasto', 'ordinario', 'pizzicato', 'ponticello', 'sul ponticello', 'sul tasto', 'col legno tratto', 'd.p.', 'm.s.p', 'm.s.t.', 'ord.', 'pizz.', 'p.', 's.p.', 's.t.', 'c.l.t.']]*) –

`contact_point: Optional[Literal['dietro ponticello', 'molto sul ponticello', 'molto sul tasto', 'ordinario', 'pizzicato', 'ponticello', 'sul ponticello', 'sul tasto', 'col legno tratto', 'd.p.', 'm.s.p', 'm.s.t.', 'ord.', 'pizz.', 'p.', 's.p.', 's.t.', 'c.l.t.']] = None`

class `Ornamentation`(*direction: Optional[Literal['up', 'down']] = None, n_times: int = 1*)

Bases: *ImplicitPlayingIndicator*

Parameters

- `direction` (*Optional[Literal['up', 'down']]*) –

- `n_times` (*int*) –

`direction: Optional[Literal['up', 'down']] = None`

`n_times: int = 1`

`class BendAfter`(*bend_amount: Optional[float] = None, minimum_length: Optional[float] = 3, thickness: Optional[float] = 3*)

Bases: *ImplicitPlayingIndicator*

Parameters

- `bend_amount` (*Optional[float]*) –
- `minimum_length` (*Optional[float]*) –
- `thickness` (*Optional[float]*) –

`bend_amount: Optional[float] = None`

`minimum_length: Optional[float] = 3`

`thickness: Optional[float] = 3`

`class ArtificalHarmonic`(*n_semitones: Optional[int] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`n_semitones` (*Optional[int]*) –

`n_semitones: Optional[int] = None`

`class PreciseNaturalHarmonic`(*string_pitch: Optional[*mutwo.music_parameters.pitches.WesternPitch.WesternPitch*] = None, played_pitch: Optional[*mutwo.music_parameters.pitches.WesternPitch.WesternPitch*] = None, harmonic_note_head_style: bool = True, parenthesize_lower_note_head: bool = False*)

Bases: *ImplicitPlayingIndicator*

Parameters

- `string_pitch` (*Optional[WesternPitch]*) –
- `played_pitch` (*Optional[WesternPitch]*) –
- `harmonic_note_head_style` (*bool*) –
- `parenthesize_lower_note_head` (*bool*) –

`harmonic_note_head_style: bool = True`

`parenthesize_lower_note_head: bool = False`

`played_pitch: Optional[WesternPitch] = None`

`string_pitch: Optional[WesternPitch] = None`

`class Fermata`(*fermata_type: Optional[Literal['shortfermata', 'fermata', 'longfermata', 'verylongfermata']] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

`fermata_type` (*Optional[Literal['shortfermata', 'fermata', 'longfermata', 'verylongfermata']]*) –

`fermata_type: Optional[Literal['shortfermata', 'fermata', 'longfermata', 'verylongfermata']] = None`

`class Hairpin`(*symbol: Optional[Literal['<', '>', '<>', '!']] = None, niente: bool = False*)

Bases: *ImplicitPlayingIndicator*

Parameters

- `symbol` (*Optional[Literal['<', '>', '<>', '!']]*) –
- `niente` (*bool*) –

`niente: bool = False`

`symbol: Optional[Literal['<', '>', '<>', '!']] = None`

class Trill(*pitch: Optional[mutwo.music_parameters.abc.Pitch] = None*)

Bases: *ImplicitPlayingIndicator*

Parameters

pitch (*Optional[Pitch]*) –

`pitch: Optional[Pitch] = None`

class WoodwindFingering(*cc: Optional[Tuple[str, ...]] = None, left_hand: Optional[Tuple[str, ...]] = None, right_hand: Optional[Tuple[str, ...]] = None, instrument: str = 'clarinet'*)

Bases: *ImplicitPlayingIndicator*

Parameters

- **cc** (*Optional[Tuple[str, ...]]*) –
- **left_hand** (*Optional[Tuple[str, ...]]*) –
- **right_hand** (*Optional[Tuple[str, ...]]*) –
- **instrument** (*str*) –

`cc: Optional[Tuple[str, ...]] = None`

`instrument: str = 'clarinet'`

`left_hand: Optional[Tuple[str, ...]] = None`

`right_hand: Optional[Tuple[str, ...]] = None`

class Cue(*cue_count=None*)

Bases: *ImplicitPlayingIndicator*

Cue for electronics etc.

Parameters

cue_count (*Optional[int]*) –

`cue_count: Optional[int] = None`

class PlayingIndicatorCollection(*articulation: mutwo.music_parameters.playing_indicators.Articulation = <factory>, artifical_harmonic: mutwo.music_parameters.playing_indicators.ArtificalHarmonic = <factory>, arpeggio: mutwo.music_parameters.playing_indicators.Arpeggio = <factory>, bartok_pizzicato: mutwo.music_parameters.abc.PlayingIndicator = <factory>, bend_after: mutwo.music_parameters.playing_indicators.BendAfter = <factory>, breath_mark: mutwo.music_parameters.abc.PlayingIndicator = <factory>, cue: mutwo.music_parameters.playing_indicators.Cue = <factory>, duration_line_dashed: mutwo.music_parameters.abc.PlayingIndicator = <factory>, duration_line_triller: mutwo.music_parameters.abc.PlayingIndicator = <factory>, fermata: mutwo.music_parameters.playing_indicators.Fermata = <factory>, glissando: mutwo.music_parameters.abc.PlayingIndicator = <factory>, hairpin: mutwo.music_parameters.playing_indicators.Hairpin = <factory>, natural_harmonic: mutwo.music_parameters.abc.PlayingIndicator = <factory>, laissez_vibrer: mutwo.music_parameters.abc.PlayingIndicator = <factory>, ornamentation: mutwo.music_parameters.playing_indicators.Ornamentation = <factory>, pedal: mutwo.music_parameters.playing_indicators.Pedal = <factory>, prall: mutwo.music_parameters.abc.PlayingIndicator = <factory>, precise_natural_harmonic: mutwo.music_parameters.playing_indicators.PreciseNaturalHarmonic = <factory>, string_contact_point: mutwo.music_parameters.playing_indicators.StringContactPoint = <factory>, tie: mutwo.music_parameters.abc.PlayingIndicator = <factory>, tremolo: mutwo.music_parameters.playing_indicators.Tremolo = <factory>, trill: mutwo.music_parameters.playing_indicators.Trill = <factory>, woodwind_fingering: mutwo.music_parameters.playing_indicators.WoodwindFingering = <factory>*)

Bases: *IndicatorCollection[PlayingIndicator]*

Parameters

- **articulation** (Articulation) –
- **artifical_harmonic** (ArtificalHarmonic) –
- **arpeggio** (Arpeggio) –

- **bartok_pizzicato** (PlayingIndicator) –
- **bend_after** (BendAfter) –
- **breath_mark** (PlayingIndicator) –
- **cue** (Cue) –
- **duration_line_dashed** (PlayingIndicator) –
- **duration_line_triller** (PlayingIndicator) –
- **fermata** (Fermata) –
- **glissando** (PlayingIndicator) –
- **hairpin** (Hairpin) –
- **natural_harmonic** (PlayingIndicator) –
- **laissez_vibrer** (PlayingIndicator) –
- **ornamentation** (Ornamentation) –
- **pedal** (Pedal) –
- **prall** (PlayingIndicator) –
- **precise_natural_harmonic** (PreciseNaturalHarmonic) –
- **string_contact_point** (StringContactPoint) –
- **tie** (PlayingIndicator) –
- **tremolo** (Tremolo) –
- **trill** (Trill) –
- **woodwind_fingering** (WoodwindFingering) –

arpeggio: *Arpeggio*

articulation: *Articulation*

artifical_harmonic: *ArtificalHarmonic*

bartok_pizzicato: *PlayingIndicator*

bend_after: *BendAfter*

breath_mark: *PlayingIndicator*

cue: *Cue*

duration_line_dashed: *PlayingIndicator*

duration_line_triller: *PlayingIndicator*

fermata: *Fermata*

glissando: *PlayingIndicator*

hairpin: *Hairpin*

laissez_vibrer: *PlayingIndicator*

natural_harmonic: *PlayingIndicator*

ornamentation: *Ornamentation*

pedal: *Pedal*

prall: *PlayingIndicator*

precise_natural_harmonic: *PreciseNaturalHarmonic*

string_contact_point: *StringContactPoint* `tie: PlayingIndicator`

`tremolo: Tremolo`

`trill: Trill`

`woodwind_fingering: WoodwindFingering`

mutwo.music_parameters.abc

Abstract base classes for different parameters.

This module defines the public API of parameters. Most other mutwo classes rely on this API. This means when someone creates a new class inheriting from any of the abstract parameter classes which are defined in this module, she or he can make use of all other mutwo modules with this newly created parameter class.

`class ExplicitPlayingIndicator(`*is_active=False*`)`

Bases: *PlayingIndicator*

Parameters

`is_active` (*bool*) –

`get_arguments_dict()`

Return type

dict[str, *Any*]

`property is_active: bool`

`class ImplicitPlayingIndicator`

Bases: *PlayingIndicator*

`property is_active: bool`

`class Indicator`

Bases: `ABC`

`get_arguments_dict()`

Return type

dict[str, *Any*]

`abstract property is_active: bool`

`class IndicatorCollection`

Bases: `Generic[T]`

`get_all_indicator()`

Return type

tuple[~T, …]

`get_indicator_dict()`

Return type

dict[str, *mutwo.music_parameters.abc.Indicator*]

`class Lyric`

Bases: *SingleValueParameter*

Abstract base class for any spoken, sung or written text.

If the user wants to define a new lyric class, the abstract properties *phonetic_representation* and *written_representation* have to be overridden.

The *phonetic_representation* should return a string of X-SAMPA format phonemes, separated by space to indicate new words. Consult wikipedia entry for detailed information regarding X-SAMPA.

The *written_representation* should return a string of normal written text, separated by space to indicate new words.

abstract property phonetic_representation: value_return_type

property value_name

property written_representation: str

Get text as it would be written in natural language

class NotationIndicator

Bases: *Indicator*

Abstract base class for any notation indicator.

property is_active: bool

class Pitch(*envelope=None*)

Bases: *SingleNumberParameter*, *ParameterWithEnvelope*

Abstract base class for any pitch class.

If the user wants to define a new pitch class, the abstract property *frequency* has to be overridden. Starting from mutwo version = 0.46.0 the user will furthermore have to define an *add()* method.

Parameters

envelope (*Optional[Union[*Pitch.PitchIntervalEnvelope*, Sequence]]*) –

class PitchEnvelope(**args*, *event_to_parameter=None*, *value_to_parameter=None*, *parameter_to_value=None*, *apply_parameter_on_event=None*, ***kwargs*)

Bases: *Envelope*

Default resolution envelope class for *Pitch*

Parameters

- **event_to_parameter** (*Optional[Callable[[*core_events.abc.Event*], core_constants.ParameterType]]*) –
- **value_to_parameter** (*Optional[Callable[[core_events.Envelope.Value], core_constants.ParameterType]]*) –
- **parameter_to_value** (*Optional[Callable[[core_constants.ParameterType], core_events.Envelope.Value]]*) –
- **apply_parameter_on_event** (*Optional[Callable[[*core_events.abc.Event*, core_constants.ParameterType], None]]*) –

classmethod frequency_and_envelope_to_pitch(*frequency*, *envelope=None*)

Parameters

- **frequency** (*Union[float, Fraction, int]*) –
- **envelope** (*Optional[Union[*PitchIntervalEnvelope*, Sequence]]*) –

Return type

Pitch

class PitchIntervalEnvelope(**args*, *event_to_parameter=None*, *value_to_parameter=None*, *parameter_to_value=<function Pitch.PitchIntervalEnvelope.<lambda>>*, *apply_parameter_on_event=None*, *base_parameter_and_relative_parameter_to_absolute_parameter=None*, ***kwargs*)

Bases: *RelativeEnvelope*

Default envelope class for *Pitch*

Resolves into *Pitch.PitchEnvelope*.

Parameters

- **event_to_parameter** (*Optional[Callable[[*core_events.abc.Event*], core_constants.ParameterType]]*) –
- **value_to_parameter** (*Optional[Callable[[core_events.Envelope.Value], core_constants.ParameterType]]*) –
- **parameter_to_value** (*Callable[[core_constants.ParameterType], core_events.Envelope.Value]*) –
- **apply_parameter_on_event** (*Optional[Callable[[*core_events.abc.Event*, core_constants.ParameterType], None]]*) –

- **base_parameter_and_relative_parameter_to_absolute_parameter** (*Optional[Callable[[core_constants.ParameterType, core_constants.ParameterType], core_constants.ParameterType]]*) –

classmethod cents_to_pitch_interval(*cents*)

Parameters
cents (*Union[float, Fraction, int]*) –

Return type
PitchInterval

abstract add(*pitch_interval*, *mutate=True*)

Parameters

- **pitch_interval** (PitchInterval) –
- **mutate** (*bool*) –

Return type
Pitch

static cents_to_ratio(*cents*)

Converts a cent value to its respective frequency ratio.

Parameters
cents (*Union[float, Fraction, int]*) – Cents that shall be converted to a frequency ratio.

Return type
Fraction

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Pitch.cents_to_ratio(1200)
Fraction(2, 1)
```

get_pitch_interval(*pitch_to_compare*)

Get *PitchInterval* between itself and other pitch

Parameters
pitch_to_compare (Pitch) – The pitch which shall be compared to the active pitch.

Returns
PitchInterval between

Return type
PitchInterval

Example:

```
>>> from mutwo import music_parameters
>>> a4 = music_parameters.DirectPitch(frequency=440)
>>> a5 = music_parameters.DirectPitch(frequency=880)
>>> a4.get_pitch_interval(a5)
DirectPitchInterval(cents = 1200)
```

static hertz_to_cents(*frequency0*, *frequency1*)

Calculates the difference in cents between two frequencies.

Parameters

- **frequency0** (*Union[float, Fraction, int]*) – The first frequency in Hertz.
- **frequency1** (*Union[float, Fraction, int]*) – The second frequency in Hertz.

Returns
The difference in cents between the first and the second frequency.

Return type
float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Pitch.hertz_to_cents(200, 400)
1200.0
```

static hertz_to_midi_pitch_number(*frequency*)

Converts a frequency in hertz to its respective midi pitch.

Parameters
`frequency` (*Union[float, Fraction, int]*) – The frequency that shall be translated to a midi pitch number.

Returns
The midi pitch number (potentially a floating point number if the entered frequency isn't on the grid of the equal divided octave tuning with a = 440 Hertz).

Return type
float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Pitch.hertz_to_midi_pitch_number(440)
69.0
>>> abc.Pitch.hertz_to_midi_pitch_number(440 * 3 / 2)
75.98044999134612
```

static ratio_to_cents(*ratio*)

Converts a frequency ratio to its respective cent value.

Parameters
`ratio` (*Fraction*) – The frequency ratio which cent value shall be calculated.

Return type
float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Pitch.ratio_to_cents(fractions.Fraction(3, 2))
701.9550008653874
```

resolve_envelope(*duration*, *resolve_envelope_class=None*)

Parameters

- `duration` (*Union[float, Fraction, int]*) –
- `resolve_envelope_class` (*Optional[type[mutwo.core_events.envelopes.Envelope]]*) –

Return type
Envelope

subtract(*pitch_interval*)

Parameters
`pitch_interval` (PitchInterval) –

Return type
Pitch

property envelope: *RelativeEnvelope*

abstract property frequency: value_return_type

property midi_pitch_number: float

The midi pitch number (from 0 to 127) of the pitch.

property value_name

class PitchAmbitus(*minima_pitch*, *maxima_pitch*)

Bases: `ABC`

Abstract base class for all pitch ambituses.

To setup a new PitchAmbitus class override the abstract method *pitch_to_period*.

Parameters

- **minima_pitch** (Pitch) –
- **maxima_pitch** (Pitch) –

filter_pitch_sequence(*pitch_to_filter_sequence*)

Filter all pitches in a sequence which aren't inside the ambitus.

Parameters

pitch_to_filter_sequence (*Sequence[Pitch]*) – A sequence with pitches which shall be filtered.

Return type

tuple[*mutwo.music_parameters.abc.Pitch*, …]

Example:

```
>>> from mutwo import music_parameters
>>> ambitus0 = music_parameters.OctaveAmbitus(
        music_parameters.JustIntonationPitch('1/2'),
        music_parameters.JustIntonationPitch('2/1'),
    )
>>> ambitus0.filter_pitch_sequence(
         [
             music_parameters.JustIntonationPitch("3/8"),
             music_parameters.JustIntonationPitch("3/4"),
             music_parameters.JustIntonationPitch("3/2"),
             music_parameters.JustIntonationPitch("3/1"),
         ]
    )
(JustIntonationPitch('3/4'), JustIntonationPitch('3/2'))
```

get_pitch_variant_tuple(*pitch*, *period=None*)

Find all pitch variants (in all octaves) of the given pitch

Parameters

- **pitch** (Pitch) – The pitch which variants shall be found.
- **period** (*Optional[PitchInterval]*) – The repeating period (usually an octave). If the period is set to *None* the function will fallback to them objects method **:method:`pitch_to_period`**. Default to *None*.

Return type

tuple[*mutwo.music_parameters.abc.Pitch*, …]

abstract pitch_to_period(*pitch*)

Parameters

pitch (Pitch) –

Return type

PitchInterval

property border_tuple: tuple[*mutwo.music_parameters.abc.Pitch*, *mutwo.music_parameters.abc.Pitch*]

property range: *PitchInterval*

class PitchInterval

Bases: *SingleNumberParameter*

Abstract base class for any pitch interval class

If the user wants to define a new pitch interval class, the abstract property *interval* has to be overridden.

interval is stored in unit *cents*.

See wikipedia entry for definition of 'cents'.

abstract property interval: value_return_type

property value_name

class PlayingIndicator

Bases: *Indicator*

Abstract base class for any playing indicator.

class Syllable(*is_last_syllable*)

Bases: *Lyric*

Syllable mixin for classes which inherit from *Lyric*.

This adds the new attribute `is_last_syllable`. This should be *True* if it is the last syllable of a word and *False* if it isn't.

Parameters

is_last_syllable (*bool*) –

class Volume

Bases: *SingleNumberParameter*

Abstract base class for any volume class.

If the user wants to define a new volume class, the abstract property *amplitude* has to be overridden.

static amplitude_ratio_to_decibel(*amplitude*, *reference_amplitude=1*)

Convert amplitude ratio to decibel.

Parameters

- **amplitude** (*Union[float, Fraction, int]*) – The amplitude that shall be converted.
- **reference_amplitude** (*Union[float, Fraction, int]*) – The amplitude for decibel == 0.

Return type

float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.amplitude_ratio_to_decibel(1)
0
>>> abc.Volume.amplitude_ratio_to_decibel(0)
inf
>>> abc.Volume.amplitude_ratio_to_decibel(0.5)
-6.020599913279624
```

static amplitude_ratio_to_midi_velocity(*amplitude*, *reference_amplitude=1*)

Convert amplitude ratio to midi velocity.

Parameters

- **amplitude** (*core_constants.Real*) – The amplitude which shall be converted.
- **reference_amplitude** (*Union[float, Fraction, int]*) – The amplitude for decibel == 0.

Returns

The midi velocity.

Return type

int

The method clips values that are higher than 1 / lower than 0.

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.amplitude_ratio_to_midi_velocity(1)
127
>>> abc.Volume.amplitude_ratio_to_midi_velocity(0)
0
```

static decibel_to_amplitude_ratio(*decibel*, *reference_amplitude=1*)

Convert decibel to amplitude ratio.

Parameters

- **decibel** (*Union[float, Fraction, int]*) – The decibel number that shall be converted.
- **reference_amplitude** (*Union[float, Fraction, int]*) – The amplitude for decibel == 0.

Return type

float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.decibel_to_amplitude_ratio(0)
1
>>> abc.Volume.decibel_to_amplitude_ratio(-6)
0.5011872336272722
>>> abc.Volume.decibel_to_amplitude_ratio(0, reference_amplitude=0.25)
0.25
```

static decibel_to_midi_velocity(*decibel_to_convert*, *minimum_decibel=None*, *maximum_decibel=None*)

Convert decibel to midi velocity (0 to 127).

Parameters

- **decibel** (*core_constants.Real*) – The decibel value which shall be converted..
- **minimum_decibel** (*core_constants.Real, optional*) – The decibel value which is equal to the lowest midi velocity (0).
- **maximum_decibel** (*core_constants.Real, optional*) – The decibel value which is equal to the highest midi velocity (127).
- **decibel_to_convert** (*Union[float, Fraction, int]*) –

Returns

The midi velocity.

Return type

int

The method clips values which are higher than 'maximum_decibel' and lower than 'minimum_decibel'.

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.decibel_to_midi_velocity(0)
127
>>> abc.Volume.decibel_to_midi_velocity(-40)
0
```

static decibel_to_power_ratio(*decibel*, *reference_amplitude=1*)

Convert decibel to power ratio.

Parameters

- **decibel** (*Union[float, Fraction, int]*) – The decibel number that shall be converted.
- **reference_amplitude** (*Union[float, Fraction, int]*) – The amplitude for decibel == 0.

Return type

float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.decibel_to_power_ratio(0)
1
>>> abc.Volume.decibel_to_power_ratio(-6)
0.251188643150958
```

(continues on next page)

(continued from previous page)

```
>>> abc.Volume.decibel_to_power_ratio(0, reference_amplitude=0.25)
0.25
```

static power_ratio_to_decibel(*amplitude*, *reference_amplitude=1*)

Convert power ratio to decibel.

Parameters

- **amplitude** (*Union[float, Fraction, int]*) – The amplitude that shall be converted.
- **reference_amplitude** (*Union[float, Fraction, int]*) – The amplitude for decibel == 0.

Return type

float

Example:

```
>>> from mutwo.parameters import abc
>>> abc.Volume.power_ratio_to_decibel(1)
0
>>> abc.Volume.power_ratio_to_decibel(0)
inf
>>> abc.Volume.power_ratio_to_decibel(0.5)
-3.010299956639812
```

abstract property amplitude: value_return_type

property decibel: Union[float, Fraction, int]

The decibel of the volume (from -120 to 0)

property midi_velocity: int

The velocity of the volume (from 0 to 127).

property value_name

mutwo.music_parameters.configurations

mutwo.music_parameters.constants

mutwo.music_utilities

Table of content

Object	Documentation
mutwo.music_utilities.DuplicatePlayingIndicatorConverterMappingWarning	

class DuplicatePlayingIndicatorConverterMappingWarning(*articulation_name*, *playing_indicator_converter*)

Bases: `RuntimeWarning`

Parameters

articulation_name (*str*) –

mutwo.music_version

Table of content

`VERSION = '0.17.1'`

The version of the package `mutwo.music`.

mutwo.reaper_converters

Table of content

Object	Documentation
mutwo.reaper_converters.ReaperMarkerConverter	Make Reaper Marker entries.

`class ReaperMarkerConverter`(*simple_event_to_marker_name=<function ReaperMarkerConverter.<lambda>>*, *simple_event_to_marker_color=<function ReaperMarkerConverter.<lambda>>*)

Bases: *EventConverter*

Make Reaper Marker entries.

param simple_event_to_marker_name
A function which converts a *SimpleEvent* to the marker name. By default the function will ask the event for its *name* property. If the event doesn't know the *name* property (and the function call will result in an `AttributeError`) mutwo will ignore the current event.

type simple_event_to_marker_name
typing.Callable[[core_events.SimpleEvent], str]

param simple_event_to_marker_color
A function which converts a *SimpleEvent* to the marker color. By default the function will ask the event for its *color* property. If the event doesn't know the *color* property (and the function call will result in an `AttributeError`) mutwo will ignore the current event.

type simple_event_to_marker_color
typing.Callable[[core_events.SimpleEvent], str]

The resulting string can be copied into the respective reaper project file one line before the '<PROJBAY' tag.

Example:

```
>>> from mutwo import reaper_converters
>>> from mutwo import core_events
>>> marker_converter = reaper_converters.ReaperMarkerConverter()
>>> events = core_events.SequentialEvent([core_events.SimpleEvent(2), core_events.SimpleEvent(3)])
>>> events[0].name = 'beginning'
>>> events[0].color = r'0 16797088 1 B {A4376701-5AA5-246B-900B-28ABC969123A}'
>>> events[1].name = 'center'
>>> events[1].color = r'0 18849803 1 B {E4DD7D23-98F4-CA97-8587-F4259A9498F7}'
>>> marker_converter.convert(events)
'MARKER 0 0 beginning 0 16797088 1 B {A4376701-5AA5-246B-900B-28ABC969123A}
```

MARKER 1 2 center 0 18849803 1 B {E4DD7D23-98F4-CA97-8587-F4259A9498F7}'

Parameters

- **`simple_event_to_marker_name`** (*`Callable[[SimpleEvent], str]`*) –
- **`simple_event_to_marker_color`** (*`Callable[[SimpleEvent], str]`*) –

`convert`(*event_to_convert*)

Convert event to reaper markers (as plain string).

Parameters

`event_to_convert` (*`events.abc.Event`*) – The event which shall be converted to reaper marker entries.

Returns

The reaper marker entries as plain strings. Copy them to your reaper project file one line before the '<PROJBAY' tag and the next time when you open the project they will appear.

Return type

str

Return type

str

mutwo.reaper_version

Table of content

`VERSION = '0.3.1'`

The version of the package `mutwo.reaper`.

PYTHON MODULE INDEX