partitura

Release 0.3.5

Nov 09, 2019

Contents

CHAPTER 1

Introduction

The principal aim of the *partitura* package is to handle richly structured musical information as conveyed by modern staff music notation. It provides a much wider range of possibilities to deal with music than the more reductive (but very common) pianoroll-oriented approach inspired by the MIDI standard.

Specifically, the package allows for representing a variety of information in musical scores beyond the onset, duration and MIDI pitch numbers of notes, such as:

- pitch spellings,
- symbolic duration categories,
- and voicing information.

Moreover, it supports musical notions that are not note-related, like:

- measures,
- tempo indications,
- performance directions,
- repeat structures,
- and time/key signatures.

In addition to score information, the package can load MIDI recordings of performed scores, and alignments between scores and performances in the *Matchfile* format used by the publicly available Vienna4x22 piano corpus research dataset.

1.1 Supported file types

Musical data can be loaded from and saved to *MusicXML* and *MIDI* files. Furthermore, *partitura* uses MuseScore as a backend to load files in other formats, like *MuseScore*, *MuseData*, and *GuitarPro*. This requires a working installation of MuseScore on your computer.

1.2 Conceptual Overview

This section offers some conceptual and design considerations that may be helpful when working with the package.

1.2.1 Representing score information

The package defines a musical ontology to describe musical scores that roughly follows the elements defined by the MusicXML specification. More specifically, the elements of a musical score are represented as a collection of instances of classes like *Note*, *Measure*, *Slur*, and *Rest*. These instances are attached to an instance of class *Part*, which corresponds to the role of an instrument in a musical score. A part may contain one or more staffs, depending on the instrument.

In contrast to MusicXML documents, where musical time is largely implicit, time plays a crucial role in the representation of scores in *partitura*. Musical elements are associated to a *Part* instance by specifying their *start* (and possibly *end*) times. The *Part* instance thus acts as a timeline consisting of a number of discrete timepoints, each of which holds references to the musical elements starting and ending at that time. The musical elements themselves contain references to their respective starting and ending timepoints. Other than that, cross-references between musical elements are used sparingly, to keep the API simple.

Musical elements in a *Part* can be filtered by class and iterated over, either from a particular timepoint onward or backward, or within a specified range. For example to find the measure to which a note belongs, you would iterate backwards over elements of class Measure that start at or before the start time of the note and select the first element of that iteration.

1.2.2 Score vs. performance

Although the MIDI format can be used to represent both score-related (key/time signatures, tempo) and performance-related information (expressive timing, dynamics), partitura regards a MIDI file as a representation of either a a score or a performance. Therefore is has separate functions to load and save scores (`load_score_midi()`, `save_score_midi()`) and performances (`load_performance_midi()`, `save_performance_midi()`). `load_score_midi()` offers simple quantization for unquantized MIDIs but in general you should not expect a MIDI representation of a performance to be loaded correctly as a *Part* instance.

1.3 Relation to music21

The *music21* package has been around since 2008, and is one of the few python packages available for working with symbolic musical data. It is both more mature and more elaborate than *partitura*. The aims of *partitura* are different from and more modest than those of *music21*, which aims to provide a toolkit for computer-aided musicology. Instead, *partitura* intends to provide a convenient way to work with symbolic musical data in the context of problems such as musical expression modeling, or music generation. Although it is not the main aim of the package to provide music analysis tools, the package does offer functionality for pitch spelling, voice assignment and key estimation.

CHAPTER 2

Usage

In this Section we will demonstrate basic usage of the package.

2.1 Importing MusicXML

We start by loading a score from a MusicXML file. As an example we take a MusicXML file with the following contents:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE score-partwise PUBLIC
  "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P1">
      <part-name>Piano</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <!--=========================================================-->
    <measure number="1">
      <attributes>
        <divisions>12</divisions>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
      </attributes>
      <print new-page="yes" new-system="yes"/>
      <note id="n01">
        <pitch>
          <step>A</step>
          <octave>4</octave>
```

(continues on next page)

(continued from previous page)

```
        </pitch>
        <duration>48</duration>
        <voice>1</voice>
        <type>whole</type>
        <staff>2</staff>
      </note>
      <backup>
        <duration>48</duration>
      </backup>
      <note id="r01">
        <rest/>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
      <note id="n02">
        <pitch>
          <step>C</step>
          <octave>5</octave>
        </pitch>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
      <note id="n03">
        <chord/>
        <pitch>
          <step>E</step>
          <octave>5</octave>
        </pitch>
        <duration>24</duration>
        <voice>2</voice>
        <type>half</type>
        <staff>1</staff>
      </note>
    </measure>
  </part>
</score-partwise>
```

To load the score in python we first import the partitura package:

```
>>> import partitura
```

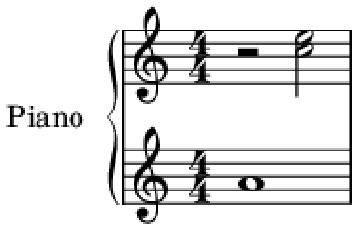
For convenience a MusicXML file with the above contents is included in the package. The path to the file is stored as *partitura.EXAMPLE_MUSICXML*, so that we load the above score as follows:

```
>>> my_musicxml_file = partitura.EXAMPLE_MUSICXML
>>> part = partitura.load_musicxml(my_musicxml_file)
```

2.2 Displaying the typeset part

The *partitura.render()* function displays the part as a typeset score:

```
>>> partitura.render(part)
```



This should open an image of the score in the default image viewing application of your desktop. The function requires that either MuseScore or lilypond is installed on your computer.

2.3 Exporting a score to MusicXML

The `partitura.save_musicxml()` function exports score information to MusicXML. The following line saves *part* to a file *mypart.musicxml*:

```
>>> partitura.save_musicxml(part, 'mypart.musicxml')
```

2.4 Viewing the musical elements

The function `load_musicxml()` returns the score as a `Part` instance. When we print it, it displays its id and part-name:

```
>>> print(part)
Part id="P1" name="Piano"
```

To see all of the elements in the part at once, we can call its `pretty()` method:

```
>>> print(part.pretty())
Part id="P1" name="Piano"
 │
 ├─ TimePoint t=0 quarter=12
 │   │
 │   └─ starting objects
 │       │
 │       ├─ Measure number=1
 │       ├─ Note id=n01 voice=1 staff=2 type=whole pitch=A4
 │       ├─ Page number=1
 │       ├─ Rest id=r01 voice=2 staff=1 type=half
 │       ├─ System number=1
 │       └─ TimeSignature 4/4
 │
 ├─ TimePoint t=24 quarter=12
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   └─ Rest id=r01 voice=2 staff=1 type=half
 │   │
 │   └─ starting objects
 │       │
```

(continues on next page)

(continued from previous page)

```
│         ├─ Note id=n02 voice=2 staff=1 type=half pitch=C5
│         └─ Note id=n03 voice=2 staff=1 type=half pitch=E5
│
└─ TimePoint t=48 quarter=12
       │
       └─ ending objects
          │
          ├─ Measure number=1
          ├─ Note id=n01 voice=1 staff=2 type=whole pitch=A4
          ├─ Note id=n02 voice=2 staff=1 type=half pitch=C5
          └─ Note id=n03 voice=2 staff=1 type=half pitch=E5
```

This reveals that the part has three time points at which one or more musical objects start or end. At *t=0* there are several starting objects, including a `TimeSignature`, `Measure`, `Page`, and `System`.

2.5 Extracting a piano roll

The notes in this part can be accessed through the property `part.notes`:

```
>>> part.notes
[<partitura.score.Note object at 0x...>, <partitura.score.Note object at 0x...>,
<partitura.score.Note object at 0x...>]
```

To create a piano roll extract from the part as a numpy array you would do the following:

```
>>> import numpy as np
>>> pianoroll = np.array([(n.start.t, n.end.t, n.midi_pitch) for n in part.notes])
>>> print(pianoroll)
[[ 0 48 69]
 [24 48 72]
 [24 48 76]]
```

The note start and end times are in the units specificied by the *divisions* element of the MusicXML file. This element specifies the duration of a quarter note. The *divisions* value can vary within an MusicXML file, so it is generally better to work with musical time in beats.

The part object has a property `part.beat_map` that converts timeline times into beat times:

```
>>> beat_map = part.beat_map
>>> print(beat_map(pianoroll[:, 0]))
[0. 2. 2.]
>>> print(beat_map(pianoroll[:, 1]))
[4. 4. 4.]
```

2.6 Iterating over arbitrary musical objects

In the previous Section we used `part.notes` to obtain the notes in the part as a list. This property is a short cut for the following statement:

```
>>> list(part.iter_all(partitura.score.Note))
[<partitura.score.Note object at 0x...>, <partitura.score.Note object at 0x...>,
<partitura.score.Note object at 0x...>]
```

Here we access the `iter_all()` method. Given a class, it iterates over all instances of that class that occur in the part:

```
>>> for m in part.iter_all(partitura.score.Measure):
...     print(m)
Measure number=1
```

The `iter_all()` method has a keyword *include_subclasses* that indicates that we are also interested in any subclasses of the specified class. For example, the following statement iterates over all objects in the part:

```
>>> for m in part.iter_all(object, include_subclasses=True):
...     print(m)
Page number=1
System number=1
Measure number=1
TimeSignature 4/4
Note id=n01 voice=1 staff=2 type=whole pitch=A4
Rest id=r01 voice=2 staff=1 type=half
Note id=n02 voice=2 staff=1 type=half pitch=C5
Note id=n03 voice=2 staff=1 type=half pitch=E5
```

This approach is useful for example when we want to retrieve rests in addition to notes. Since rests and notes are both subclassess of `GenericNote`, the following works:

```
>>> for m in part.iter_all(partitura.score.GenericNote, include_subclasses=True):
...     print(m)
Note id=n01 voice=1 staff=2 type=whole pitch=A4
Rest id=r01 voice=2 staff=1 type=half
Note id=n02 voice=2 staff=1 type=half pitch=C5
Note id=n03 voice=2 staff=1 type=half pitch=E5
```

By default, *include_subclasses* is False.

2.7 Creating a musical score by hand

You can build a musical score from scratch, by creating a *Part* object. We start by renaming the *partitura.score* module to *score*, for convenience:

```
>>> import partitura.score as score
```

Then we create an empty part with id 'P0' and name 'My Part' (the name is optional, the id is mandatory), and at t=0 specify that a quarter note duration equals a time interval of 10.

```
>>> part = score.Part('P0', 'My Part')
>>> part.set_quarter_duration(0, 10)
```

Adding elements to the part is done by the `add()` method, which takes a musical element, a start and an end time. Either of the *start* and *end* arguments can be omitted, but if both are omitted the method will do nothing.

We now add a 3/4 time signature at t=0, and three notes. The notes are instantiated by specifying an (optional) id, pitch information, and an (optional) voice:

```
>>> part.add(score.TimeSignature(3, 4), start=0)
>>> part.add(score.Note(id='n0', step='A', octave=4, voice=1), start=0, end=10)
```

(continues on next page)

(continued from previous page)

```
>>> part.add(score.Note(id='n1', step='C', octave=5, alter=1, voice=2), start=0,␣
↪end=10)
>>> part.add(score.Note(id='n2', step='C', octave=5, alter=1, voice=2), start=10,␣
↪end=40)
```

Note that the duration of notes is not hard-coded in the Note instances, but defined implicitly by their start and end times in the part.

Here's what the part looks like:

```
>>> print(part.pretty())
Part id="P0" name="My Part"
 │
 ├─ TimePoint t=0 quarter=10
 │   │
 │   └─ starting objects
 │       │
 │       ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │       ├─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │       └─ TimeSignature 3/4
 │
 ├─ TimePoint t=10 quarter=10
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │   │   └─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │   │
 │   └─ starting objects
 │       │
 │       └─ Note id=n2 voice=2 staff=None type=half. pitch=C#5
 │
 └─ TimePoint t=40 quarter=10
     │
     └─ ending objects
         │
         └─ Note id=n2 voice=2 staff=None type=half. pitch=C#5
```

We see that the notes n0, n1, and n2 have been correctly recognized as quarter, quarter, and dotted half, respectively.

Let's save the part to MusicXML:

```
>>> partitura.save_musicxml(part, 'mypart.musicxml')
```

When we look at the contents of *mypart.musicxml*, surprisinly, the *<part></part>* element is empty:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE score-partwise PUBLIC
  "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <part-list>
    <score-part id="P0">
      <part-name>My Part</part-name>
    </score-part>
  </part-list>
  <part id="P0"/>
</score-partwise>
```

The problem with our newly created part is that it contains no measures. Since the MusicXML format requires musical elements to be contained in measures, saving the part to MusicXML omits the objects we added.

2.8 Adding measures

One option to add measures is to add them by hand like we've added the notes and time signature. A more convenient alternative is to use the function *add_measures()*:

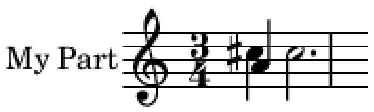
```
>>> score.add_measures(part)
```

This function uses the time signature information in the part to add measures accordingly:

```
>>> print(part.pretty())
Part id="P0" name="My Part"
 │
 ├─ TimePoint t=0 quarter=10
 │   │
 │   └─ starting objects
 │       │
 │       ├─ Measure number=1
 │       ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │       ├─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │       └─ TimeSignature 3/4
 │
 ├─ TimePoint t=10 quarter=10
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │   │   └─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │   │
 │   └─ starting objects
 │       │
 │       └─ Note id=n2 voice=2 staff=None type=half. pitch=C#5
 │
 ├─ TimePoint t=30 quarter=10
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   └─ Measure number=1
 │   │
 │   └─ starting objects
 │       │
 │       └─ Measure number=2
 │
 └─ TimePoint t=40 quarter=10
     │
     └─ ending objects
         │
         ├─ Measure number=2
         └─ Note id=n2 voice=2 staff=None type=half. pitch=C#5
```

Let' see what our part with measures looks like in typeset form:

```
>>> partitura.render(part)
```

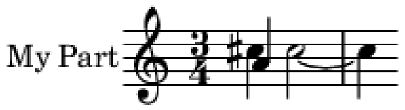


Although the notes are there, the music is not typeset correctly, since the first measure should have a duration of three quarter notes, but instead is has a duration of four quarter notes. The problem is that the note *n2* crosses a measure boundary, and thus should be tied.

2.9 Splitting up notes using ties

In musical notation notes that span measure boundaries are split up, and then tied together. This can be done automatically using the function `tie_notes()`:

```
>>> score.tie_notes(part)
>>> partitura.render(part)
```



Now the score looks correct. Displaying the contents reveals that the part now has an extra quarter note *n2a* that starts at the measure boundary, whereas the note *n2* is now a half note, ending at the measure boundary.

```
>>> print(part.pretty())
Part id="P0" name="My Part"
 │
 ├─ TimePoint t=0 quarter=10
 │   │
 │   └─ starting objects
 │       │
 │       ├─ Measure number=1
 │       ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │       ├─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │       └─ TimeSignature 3/4
 │
 ├─ TimePoint t=10 quarter=10
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   ├─ Note id=n0 voice=1 staff=None type=quarter pitch=A4
 │   │   └─ Note id=n1 voice=2 staff=None type=quarter pitch=C#5
 │   │
 │   └─ starting objects
 │       │
 │       └─ Note id=n2 voice=2 staff=None type=half tie_group=n2+n2a pitch=C#5
 │
 ├─ TimePoint t=30 quarter=10
 │   │
 │   ├─ ending objects
 │   │   │
 │   │   ├─ Measure number=1
 │   │   └─ Note id=n2 voice=2 staff=None type=half tie_group=n2+n2a pitch=C#5
 │   │
 │   └─ starting objects
 │       │
```

(continues on next page)

(continued from previous page)

```
 │         ├─ Measure number=2
 │         └─ Note id=n2a voice=2 staff=None type=quarter tie_group=n2+n2a pitch=C#5
 │
 └─ TimePoint t=40 quarter=10
        │
        └─ ending objects
            │
            ├─ Measure number=2
            └─ Note id=n2a voice=2 staff=None type=quarter tie_group=n2+n2a pitch=C#5
```

2.10 Removing elements

Just like we can add elements to a part, we can also remove them, using the `remove()` method. The following lines remove the measure instances that were added using the `add_measures()` function:

```
>>> for measure in part.iter_all(score.Measure):
...     part.remove(measure)
```

CHAPTER 3

partitura

The top level of the package contains functions to load and save data, display rendered scores, and functions to estimate pitch spelling, voice assignment, and key signature.

`partitura.`**`load_musicxml`**(*xml*, *ensure_list=False*, *validate=False*, *force_note_ids=False*)

Parse a MusicXML file and build a composite score ontology structure from it (see also scoreontology.py).

Parameters

- **`xml`** (*`str or file-like object`*) – Path to the MusicXML file to be parsed, or a file-like object
- **`ensure_list`** (*`bool, optional`*) – When True return a list independent of how many part or partgroup elements were created from the MIDI file. By default, when the return value of *load_musicxml* produces a
- **`single`** (class:*partitura.score.Part* or) – *`partitura.score.PartGroup`* element, the element itself is returned instead of a list containing the element. Defaults to False.
- **`validate`** (*`bool, optional`*) – When True the validity of the MusicXML is checked against the MusicXML 3.1 specification before loading the file. An exception will be raised when the MusicXML is invalid. Defaults to False.
- **`force_note_ids`** (*`bool, optional.`*) – When True each Note in the returned Part(s) will have a newly assigned unique id attribute. Existing note id attributes in the MusicXML will be discarded.

Returns **partlist** – A list of either Part or PartGroup objects

Return type list

`partitura.`**`save_musicxml`**(*parts*, *out=None*)

Save a one or more Part or PartGroup instances in MusicXML format.

Parameters

- **`parts`** (*`list, Part, or PartGroup`*) – A *`partitura.score.Part`* object, *`partitura.score.PartGroup`* or a list of these
- **`out`** (*`str, file-like object, or None, optional`*) – Output file

Returns If no output file is specified using *out* the function returns the MusicXML data as a string. Otherwise the function returns None.

Return type None or str

`partitura.`**`musicxml_to_notearray`**(*fn*, *flatten_parts=True*, *sort_onsets=True*, *expand_grace_notes=True*, *validate=False*, *beat_times=True*)

Return pitch, onset, and duration information for notes from a MusicXML file as a structured array.

By default a single array is returned by combining the note information of all parts in the MusicXML file.

Parameters

- **`fn`** (`str`) – Path to a MusicXML file
- **`flatten_parts`** (`bool`) – If *True*, returns a single array containing all notes. Otherwise, returns a list of arrays for each part.
- **`expand_grace_notes`** (`bool or 'delete'`) – When True, grace note onset and durations will be adjusted to have a non-zero duration.
- **`beat_times`** (`bool`) – When True (default) return onset and duration in beats. Otherwise, return the onset and duration in divisions.

Returns **score** – Structured array containing the score. The fields are 'pitch', 'onset' and 'duration'.

Return type structured array or list of structured arrays

`partitura.`**`load_score_midi`**(*fn*, *part_voice_assign_mode=0*, *ensure_list=False*, *quantization_unit=None*, *estimate_voice_info=True*, *estimate_key=False*, *assign_note_ids=True*)

Load a musical score from a MIDI file and return it as a Part instance.

This function interprets MIDI information as describing a score. Pitch names are estimated using Meredith's PS13 algorithm[1]. Assignment of notes to voices can either be done using Chew and Wu's voice separation algorithm[2], or by choosing one of the part/voice assignment modes that assign voices based on track/channel information. Furthermore, the key signature can be estimated based on Krumhansl's 1990 key profiles[3].

This function expects times to be metrical/quantized. Optionally a quantization unit may be specified. If you wish to access the non- quantized time of MIDI events you may wish to used the *load_performance_midi* function instead.

Parameters

- **`fn`** (`str`) – Path to MIDI file
- **`part_voice_assign_mode`** (`{0, 1, 2, 3, 4, 5}, optional`) – This keyword controls how part and voice information is associated to track and channel information in the MIDI file. The semantics of the modes is as follows:

  0 Return one Part per track, with voices assigned by channel

  1 Return one PartGroup per track, with Parts assigned by channel (no voices)

  2 Return single Part with voices assigned by track (tracks are combined, channel info is ignored)

  3 Return one Part per track, without voices (channel info is ignored)

  4 Return single Part without voices (channel and track info is ignored)

[1] Meredith, D. (2006). "The ps13 Pitch Spelling Algorithm". Journal of New Music Research, 35(2):121.

[2] Chew, E. and Wu, Xiaodan (2004) "Separating Voices in Polyphonic Music: A Contig Mapping Approach". In Uffe Kock, editor, Computer Music Modeling and Retrieval (CMMR), pp. 1–20, Springer Berlin Heidelberg.

[3] Krumhansl, Carol L. (1990) "Cognitive foundations of musical pitch", Oxford University Press, New York.

5 Return one Part per <track, channel> combination, without voices Defaults to 0.

- **ensure_list** (*bool, optional*) – When True, return a list independent of how many part or partgroup elements were created from the MIDI file. By default, when the return value of *load_score_midi* produces a single `partitura.score.Part` or `partitura.score.PartGroup` element, the element itself is returned instead of a list containing the element. Defaults to False.
- **quantization_unit** (*integer or None, optional*) – Quantize MIDI times to multiples of this unit. If None, the quantization unit is chosen automatically as the smallest division of the parts per quarter (MIDI "ticks") that can be represented as a symbolic duration. Defaults to None.
- **estimate_key** (*bool, optional*) – When True use Krumhansl's 1990 key profiles[3] to determine the most likely global key, discarding any key information in the MIDI file.
- **estimate_voice_info** (*bool, optional*) – When True use Chew and Wu's voice separation algorithm[2] to estimate voice information. This option is ignored for part/voice assignment modes that infer voice information from the track/channel info (i.e. *part_voice_assign_mode* equals 1, 3, 4, or 5). Defaults to True.

Returns One or more part or partgroup objects

Return type `partitura.score.Part`, `partitura.score.PartGroup`, or a list of these

References

partitura.**save_score_midi**(*parts*, *out*, *part_voice_assign_mode=0*, *velocity=64*)

Write data from Part objects to a MIDI file

Parameters

- **parts** (*Part, PartGroup or list of these*) – The musical score to be saved.
- **out** (*str or file-like object*) – Either a filename or a file-like object to write the MIDI data to.
- **part_voice_assign_mode** (*{0, 1, 2, 3, 4, 5}, optional*) – This keyword controls how part and voice information is associated to track and channel information in the MIDI file. The semantics of the modes is as follows:

  0 Write one track for each Part, with channels assigned by voices

  1 Write one track for each PartGroup, with channels assigned by Parts (voice info is lost) (There can be multiple levels of partgroups, I suggest using the highest level of partgroup/part) [note: this will e.g. lead to all strings into the same track] Each part not in a PartGroup will be assigned its own track

  2 Write a single track with channels assigned by Part (voice info is lost)

  3 Write one track per Part, and a single channel for all voices (voice info is lost)

  4 Write a single track with a single channel (Part and voice info is lost)

  5 Return one track per <Part, voice> combination, each track having a single channel.
- **velocity** (*int, optional*) – Default velocity for all MIDI notes.

partitura.**load_via_musescore**(*fn*)

Load a score through through the MuseScore program.

This function attempts to load the file in MuseScore, export it as MusicXML, and then load the MusicXML. This should enable loading of all file formats that for which MuseScore has import-support (e.g. MIDI, and ABC, but currently not MEI).

Parameters **fn** (*str*) – Filename of the score to load

Returns One or more part or partgroup objects

Return type *partitura.score.Part*, *partitura.score.PartGroup*, or a list of these

partitura.**load_performance_midi**(*fn*, *default_bpm=120*)

Load a musical performance from a MIDI file.

This function should be used for MIDI files that encode performances, such as those obtained from a capture of a MIDI instrument. This function loads note on/off events as well as control events, but ignores other data such as time and key signatures. Furthermore, the PerformedPart instance that the function returns does not retain the ticks_per_beat or tempo events. The timing of all events is represented in seconds. If you wish to retain this information consider using the *load_score_midi* function.

Parameters

- **fn** (*str*) – Path to MIDI file
- **default_bpm** (*number, optional*) – Tempo to use wherever the MIDI does not specify a tempo. Defaults to 120.

Returns A PerformedPart instance.

Return type *partitura.performance.PerformedPart*

partitura.**save_performance_midi**(*performed_part*, *out*, *mpq=500000*, *ppq=480*, *default_velocity=64*)

Save a *PerformedPart* instance as a MIDI file.

Parameters

- **performed_part** (*PerformedPart*) – The performed part to save
- **out** (*str or file-like object*) – Either a filename or a file-like object to write the MIDI data to.
- **mpq** (*int, optional*) – Microseconds per quarter note. This is known in MIDI parlance as the "tempo" value. Defaults to 500000 (i.e. 120 BPM).
- **ppq** (*int, optional*) – Parts per quarter, also known as ticks per beat. Defaults to 480.
- **default_velocity** (*int, optional*) – A default velocity value (between 0 and 127) to be used for notes without a specified velocity. Defaults to 64.

partitura.**render**(*part*, *fmt='png'*, *dpi=90*, *out_fn=None*)

Create a rendering of one or more parts or partgroups.

The function can save the rendered image to a file (when *out_fn* is specified), or shown in the default image viewer application.

Rendering is first attempted through musecore, and if that fails through lilypond. If that also fails the function returns without raising an exception.

Parameters

- **part** (*partitura.score.Part* or *partitura.score.PartGroup* or list of these) – The score content to be displayed
- **fmt** (*{'png', 'pdf'}, optional*) – The image format of the rendered material

- **out_fn** (*str or None, optional*) – The path of the image output file. If None, the rendering will be displayed in a viewer.

partitura.**EXAMPLE_MUSICXML = '/home/docs/.cache/Python-Eggs/partitura-0.3.5-py3.7.egg-tmp/p**

An example MusicXML file for didactic purposes

CHAPTER 4

partitura.score

This module defines an ontology of musical elements to represent musical scores, such as measures, notes, slurs, words, tempo and loudness directions. A score is defined at the highest level by a *Part* object (or a hierarchy of *Part* objects, in a *PartGroup* object). This object serves as a timeline at which musical elements are registered in terms of their start and end times.

class `partitura.score.`**`Part`**(*id*, *part_name=None*, *part_abbreviation=None*)

Bases: `object`

Represents a score part, e.g. all notes of one single instrument (or multiple instruments written in the same staff). Note that there may be more than one staff per score part.

Parameters

- **id** (*str*) – The identifier of the part. In order to be compatible with MusicXML the identifier should not start with a number.
- **part_name** (*str or None, optional*) – Name for the part. Defaults to None
- **part_abbreviation** (*str or None, optional*) – Abbreviated name for part

id

See parameters

Type str

part_name

See parameters

Type str

part_abbreviation

See parameters

Type str

pretty()

Return a pretty representation of this object.

Returns A pretty representation

Return type str

time_signature_map
A function mapping timeline times to the beats and beat_type of the time signature at that time. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

beat_map
A function mapping timeline times to beat times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

inv_beat_map
A function mapping beat times to timeline times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

quarter_map
A function mapping timeline times to quarter times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

inv_quarter_map
A function mapping quarter times to timeline times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

notes
Return a list of all Note objects in the part. This list includes GraceNote objects but not Rest objects.

Returns list of Note objects

Return type list

notes_tied
Return a list of all Note objects in the part that are either not tied, or the first note of a group of tied notes. This list includes GraceNote objects but not Rest objects.

Returns list of Note objects

Return type list

quarter_durations(*start=None*, *end=None*)
Return an Nx2 array with quarter duration (second column) and their respective times (first column).

When a start and or end time is specified, the returned array will contain only the entries within those bounds.

Parameters

- **start** (*number, optional*) – Start of range

- **end** (*number, optional*) – End of range

Returns An array with quarter durations and times

Return type ndarray

quarter_duration_map

A function mapping timeline times to quarter durations in effect at those times. The function can take scalar values or lists/arrays of values.

Returns The mapping function

Return type function

set_quarter_duration(*t*, *quarter*)

Set the duration of a quarter note from timepoint *t* onwards.

Setting the quarter note duration defines how intervals between timepoints are related to musical durations. For example when two timepoints *t1* and *t2* have associated times 10 and 20 respecively, then the interval between *t1* and *t2* corresponds to a half note when the quarter duration equals 5 during that interval.

The quarter duration can vary throughout the part. When setting a quarter duration at time t, then that value takes effect until the time of the next quarter duration. If a different quarter duration was already set at time t, it wil be replaced.

Note setting the quarter duration does not change the timepoints, only the relation to musical time. For illustration: in the example above, when changing the current quarter duration from 5 to 10, a note that starts at *t1* and ends at *t2* will change from being a half note to being a quarter note.

Parameters

- **t** (*int*) – Time at which to set the quarter duration
- **quarter** (*int*) – The quarter duration

get_point(*t*)

Return the *TimePoint* object with time *t*, or None if there is no such object.

get_or_add_point(*t*)

Return the *TimePoint* object with time *t*; if there is no such object, create it, add it to the time line, and return it.

Parameters **t** (*int*) – time value *t*

Returns a TimePoint object with time *t*

Return type *TimePoint*

add(*o*, *start=None*, *end=None*)

Add an object to the timeline.

An object can be added by start time, end time, or both, depending on which of the *start* and *end* keywords are provided. If neither is provided this method does nothing.

start and *end* should be non-negative integers.

Parameters

- **o** (*TimedObject*) – Object to be removed
- **start** (*int, optional*) – The start time of the object
- **end** (*int, optional*) – The end time of the object

remove (*o*, *which='both'*)

Remove an object from the timeline.

An object can be removed by start time, end time, or both.

Parameters

- **o** (*TimedObject*) – Object to be removed
- **which** (*{'start', 'end', 'both'}, optional*) – Whether to remove o as a starting object, an ending object, or both. Defaults to 'both'.

iter_all (*cls*, *start=None*, *end=None*, *include_subclasses=False*, *mode='starting'*)

Iterate (in direction of increasing time) over all instances of *cls* that either start or end (depending on *mode*) in the interval *start* to *end*. When *start* and *end* are omitted, the whole timeline is searched.

Parameters

- **cls** (*class*) – The class to search for
- **start** (*TimePoint*, optional) – The start of the interval to search. If omitted or None, the search starts at the start of the timeline. Defaults to None.
- **end** (*TimePoint*, optional) – The end of the interval to search. If omitted or None, the search ends at the end of the timeline. Defaults to None.
- **include_subclasses** (*bool, optional*) – If True also return instances that are subclasses of *cls*. Defaults to False.
- **mode** (*{'starting', 'ending'}, optional*) – Flag indicating whether to search for starting or ending objects. Defaults to 'starting'.

Yields *cls* – Instances of type *cls*

last_point

The last TimePoint on the timeline, or None if the timeline is empty.

Returns

Return type *TimePoint*

first_point

The first TimePoint on the timeline, or None if the timeline is empty.

Returns

Return type *TimePoint*

note_array

A structured array containing pitch, onset, duration, voice and id for each note

class partitura.score.**TimePoint** (*t*, *quarter=None*)

Bases: `partitura.utils.generic.ComparableMixin`

A TimePoint represents a temporal position within a *Part*.

TimePoints are used to keep track of the starting and ending of musical elements in the part. They are created automatically when adding musical elements to a part using its `add()` method, so there should be normally no reason to instantiate TimePoints manually.

Parameters

- **t** (*int*) – The time associated to this TimePoint. Should be a non- negative integer.
- **quarter** (*int*) – The duration of a quarter note at this TimePoint

`t`
See parameters

Type int

`quarter`
See parameters

Type int

`starting_objects`
A dictionary where the musical objects starting at this time are grouped by class.

Type dictionary

`ending_objects`
A dictionary where the musical objects ending at this time are grouped by class.

Type dictionary

`prev`
The preceding TimePoint (or None if there is none)

Type *TimePoint*

`next`
The succeding TimePoint (or None if there is none)

Type *TimePoint*

`add_starting_object`(*obj*)
Add object *obj* to the list of starting objects.

`remove_starting_object`(*obj*)
Remove object *obj* from the list of starting objects.

`remove_ending_object`(*obj*)
Remove object *obj* from the list of ending objects.

`add_ending_object`(*obj*)
Add object *obj* to the list of ending objects.

`iter_starting`(*cls*, *include_subclasses=False*)
Iterate over all objects of type *cls* that start at this time point.

Parameters

- **`cls`** (`class`) – The type of objects to iterate over
- **`include_subclasses`** (`bool, optional`) – When True, include all objects of all subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instance of type *cls*

`iter_ending`(*cls*, *include_subclasses=False*)
Iterate over all objects of type *cls* that end at this time point.

Parameters

- **`cls`** (`class`) – The type of objects to iterate over
- **`include_subclasses`** (`bool, optional`) – When True, include all objects of all subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instance of type *cls*

iter_prev(*cls*, *eq=False*, *include_subclasses=False*)
Iterate backwards in time from the current timepoint over starting object(s) of type *cls*.

Parameters

- **cls** (`class`) – Class of objects to iterate over
- **eq** (`bool, optional`) – If True start iterating at the current timepoint, rather than its predecessor. Defaults to False.
- **include_subclasses** (`bool, optional`) – If True include subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instances of *cls*

iter_next(*cls*, *eq=False*, *include_subclasses=False*)
Iterate forwards in time from the current timepoint over starting object(s) of type *cls*.

Parameters

- **cls** (`class`) – Class of objects to iterate over
- **eq** (`bool, optional`) – If True start iterating at the current timepoint, rather than its successor. Defaults to False.
- **include_subclasses** (`bool, optional`) – If True include subclasses of *cls* in the iteration. Defaults to False.

Yields *cls* – Instances of *cls*

class partitura.score.**TimedObject**
Bases: `partitura.utils.generic.ReplaceRefMixin`

This is the base class of all classes that have a start and end point. The start and end attributes initialized to None, and are set/unset when the object is added to/removed from a Part, using its `add()` and `remove()` methods, respectively.

start
Start time of the object

Type `TimePoint`

end
End time of the object

Type `TimePoint`

class partitura.score.**GenericNote**(*id=None*, *voice=None*, *staff=None*, *symbolic_duration=None*, *articulations=None*)
Bases: `partitura.score.TimedObject`

Represents the common aspects of notes and rests (and in the future unpitched notes)

Parameters

- **voice** (`integer, optional (default: None)`) –
- **id** (`integer, optional (default: None)`) –

symbolic_duration
The symbolic duration of the note.

This property returns a dictionary specifying the symbolic duration of the note. The dictionary may have the following keys:

- type : the note type as a string, e.g. ‘quarter’, ‘half’

- dots : an integer specifying the number of dots. When this key is missing it means there are no dots.
- actual_notes : Specifies the number of actual notes in a rhythmical tuplet. Used in conjunction with *normal_notes*.
- normal_notes : Specifies the normal number of notes in a rhythmical tuplet. For example a triplet of eights in the time of two eights would correspond to actual_notes=3, normal_notes=2.

The symbolic duration dictionary of a note can either be set manually (for example by specifying the *symbolic_duration* constructor keyword argument), or left unspecified (i.e. None). In the latter case the symbolic duration is estimated dynamically based on the note start and end times. Note that this latter case is generally preferrable because it ensures that the symbolic duration is consistent with the numeric duration.

If the symbolic duration cannot be estimated from the numeric duration None is returned.

Returns A dictionary specifying the symbolic duration of the note, or None if the symbolic duration could not be estimated from the numeric duration.

Return type dict or None

`duration`
The duration of the note in divisions

Returns

Return type int

`end_tied`
The *Timepoint* corresponding to the end of the note, or—when this note belongs to a group of tied notes—the end of the last note in the group.

Returns End of note

Return type *TimePoint*

`duration_tied`
Time difference of the start of the note to the end of the note, or—when this note belongs to a group of tied notes—the end of the last note in the group.

Returns Duration of note

Return type int

`duration_from_symbolic`
Return the numeric duration given the symbolic duration of the note and the quarter_duration in effect.

Returns

Return type int or None

`tie_prev_notes`
TODO

Returns Description of return value

Return type type

`tie_next_notes`
TODO

Returns Description of return value

Return type type

iter_chord(*same_duration=True*, *same_voice=True*)

Iterate over notes with coinciding start times.

Parameters

- **same_duration** (*bool, optional*) – When True limit the iteration to notes that have the same duration as the current note. Defaults to True.
- **same_voice** (*bool, optional*) – When True limit the iteration to notes that have the same voice as the current note. Defaults to True.

Yields *GenericNote*

class partitura.score.**Note**(*step*, *octave*, *alter=None*, **args*, ***kwargs*)

Bases: *partitura.score.GenericNote*

midi_pitch

The midi pitch value of the note (MIDI note number). C4 (middle C, in german: c') is note number 60.

Returns The note's pitch as MIDI note number.

Return type integer

alter_sign

The alteration of the note

Returns

Return type str

class partitura.score.**Rest**(**args*, ***kwargs*)

Bases: *partitura.score.GenericNote*

class partitura.score.**GraceNote**(*grace_type*, **args*, *steal_proportion=None*, ***kwargs*)

Bases: *partitura.score.Note*

iter_grace_seq(*backwards=False*)

Iterate over this and all subsequent/preceding grace notes, excluding the main note.

Parameters **backwards** (*bool, optional*) – When True, iterate over preceding grace notes. Otherwise iterate over subsequent grace notes. Defaults to False.

Yields *GraceNote*

class partitura.score.**Page**(*number=0*)

Bases: *partitura.score.TimedObject*

A page in a musical score. Its start and end times describe the range of musical time that is spanned by the page.

Parameters **number** (*int, optional*) – The number of the system. Defaults to 0.

number

See parameters

Type int

class partitura.score.**System**(*number=0*)

Bases: *partitura.score.TimedObject*

A system in a musical score. Its start and end times describe the range of musical time that is spanned by the system.

Parameters **number** (*int, optional*) – The number of the system. Defaults to 0.

number

See parameters

Type int

class `partitura.score.`**`Clef`**(*number*, *sign*, *line*, *octave_change*)

Bases: `partitura.score.TimedObject`

A clef.

Clefs associate the lines of a staff to musical pitches.

Parameters

- **`number`** (*`int, optional`*) – The number of the staff to which this clef belongs.
- **`sign`** (*`{'G', 'F', 'C', 'percussion', 'TAB', 'jianpu', 'none'}`*) – The sign of the clef
- **`line`** (*`int`*) – The staff line at which the sign is positioned
- **`octave_change`** (*`int`*) – The number of octaves to shift the pitches up (postive) or down (negative)

`nr`

See parameters

Type int

`sign`

See parameters

Type {‘G’, ‘F’, ‘C’, ‘percussion’, ‘TAB’, ‘jianpu’, ‘none’}

`line`

See parameters

Type int

`octave_change`

See parameters

Type int

class `partitura.score.`**`Slur`**(*start_note=None*, *end_note=None*)

Bases: `partitura.score.TimedObject`

A slur.

Slurs indicate musical grouping across notes.

Parameters

- **`start_note`** (`Note`, optional) – The note at which this slur starts. Defaults to None.
- **`end_note`** (`Note`, optional) – The note at which this slur ends. Defaults to None.

`start_note`

See parameters

Type `Note` or None

`end_note`

See parameters

Type `Note` or None

class `partitura.score.`**`Tuplet`**(*start_note=None*, *end_note=None*)

Bases: `partitura.score.TimedObject`

A tuplet.

Tuplets indicate musical grouping across notes.

Parameters

- **`start_note`** (*`Note`*, optional) – The note at which this tuplet starts. Defaults to None.
- **`end_note`** (*`Note`*, optional) – The note at which this tuplet ends. Defaults to None.

`start_note`
See parameters

Type *`Note`* or None

`end_note`
See parameters

Type *`Note`* or None

`class` `partitura.score.`**`Repeat`**
Bases: *`partitura.score.TimedObject`*

A repeat.

This class represents a repeated section in the score, designated by its start and end times.

`class` `partitura.score.`**`DaCapo`**
Bases: *`partitura.score.TimedObject`*

A Da Capo sign.

`class` `partitura.score.`**`Fine`**
Bases: *`partitura.score.TimedObject`*

A Fine sign.

`class` `partitura.score.`**`Fermata`**`(`*ref=None*`)`
Bases: *`partitura.score.TimedObject`*

A Fermata sign.

Parameters **`ref`** (*`TimedObject`* or None, optional) – An object to which this fermata applies. In practice this is a Note or a Barline. Defaults to None.

`ref`
See parameters

Type *`TimedObject`* or None

`class` `partitura.score.`**`Ending`**`(`*number*`)`
Bases: *`partitura.score.TimedObject`*

Class that represents one part of a 1—2— type ending of a musical passage (a.k.a Volta brackets).

Parameters **`number`** (*`int`*) – The number associated to this ending

`number`
See parameters

Type int

`class` `partitura.score.`**`Measure`**`(`*number=None*`)`
Bases: *`partitura.score.TimedObject`*

A measure.

Parameters **`number`** (*`int or None, optional`*) – The number of the measure. Defaults to None

`number`
See parameters

Type int

`page`
The page number on which this measure appears, or None if there is no associated page.

Returns

Return type int or None

`system`
The system number in which this measure appears, or None if there is no associated system.

Returns

Return type int or None

class `partitura.score.`**`TimeSignature`**(*beats*, *beat_type*)
Bases: *`partitura.score.TimedObject`*

A time signature.

Parameters

- **`beats`** (*`int`*) – The number of beats in a measure
- **`beat_type`** (*`int`*) – The note type that defines the beat unit. (4 for quarter notes, 2 for half notes, etc.)

`beats`
See parameters

Type int

`beat_type`
See parameters

Type int

class `partitura.score.`**`Tempo`**(*bpm*, *unit=None*)
Bases: *`partitura.score.TimedObject`*

A tempo indication.

Parameters

- **`bpm`** (*`number`*) – The tempo indicated in rate per minute
- **`unit`** (*`str or None, optional`*) – The unit to which the specified rate correspnds. This is a string that expreses a duration category, such as “q” for quarter “h.” for dotted half, and so on. When None, the unit is assumed to be quarters. Defaults to None.

`bpm`
See parameters

Type number

`unit`
See parameters

Type str or None

`microseconds_per_quarter`
The number of microseconds per quarter under this tempo.

This is useful for MIDI representations.

Returns

Return type int

class `partitura.score.`**`KeySignature`**(*fifths*, *mode*)
Bases: *`partitura.score.TimedObject`*

Key signature.

Parameters

- **`fifths`** (*`number`*) – Number of sharps (positive) or flats (negative)
- **`mode`** (*`str`*) – Mode of the key, either 'major' or 'minor'

`fifths`
See parameters

Type number

`mode`
See parameters

Type str

`name`
The key signature name, where the root is uppercase, and an trailing 'm' indicates minor modes (e.g. 'Am', 'G#').

Returns The key signature name

Return type str

class `partitura.score.`**`Transposition`**(*diatonic*, *chromatic*)
Bases: *`partitura.score.TimedObject`*

Represents a <transpose> tag that tells how to change all (following) pitches of that part to put it to concert pitch (i.e. sounding pitch).

Parameters

- **`diatonic`** (*`int`*) –
- **`chromatic`** (*`int`*) – The number of semi-tone steps to add or subtract to the pitch to get to the (sounding) concert pitch.

`diatonic`
See parameters

Type int

`chromatic`
See parameters

Type int

class `partitura.score.`**`Words`**(*text*, *staff=None*)
Bases: *`partitura.score.TimedObject`*

A textual element in the score.

Parameters

- **`text`** (*`str`*) – The text

- **staff** (*int or None, optional*) – The staff to which the text is associated. Defaults to None

text
See parameters

Type str

staff
See parameters

Type int or None, optional

class partitura.score.**Direction**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.TimedObject*

Base class for performance directions in the score.

class partitura.score.**LoudnessDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**TempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ArticulationDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ConstantDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**DynamicDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ImpulsiveDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.Direction*

class partitura.score.**ConstantLoudnessDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**ConstantTempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.TempoDirection*

class partitura.score.**ConstantArticulationDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantDirection*, *partitura.score.ArticulationDirection*

class partitura.score.**DynamicLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**DynamicTempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicDirection*, *partitura.score.TempoDirection*

class partitura.score.**IncreasingLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicLoudnessDirection*

class partitura.score.**DecreasingLoudnessDirection**(**args*, *wedge=False*, ***kwargs*)
Bases: *partitura.score.DynamicLoudnessDirection*

class partitura.score.**IncreasingTempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicTempoDirection*

class partitura.score.**DecreasingTempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.DynamicTempoDirection*

class partitura.score.**ImpulsiveLoudnessDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ImpulsiveDirection*, *partitura.score.LoudnessDirection*

class partitura.score.**ResetTempoDirection**(*text*, *raw_text=None*, *staff=None*)
Bases: *partitura.score.ConstantTempoDirection*

class partitura.score.**PartGroup**(*group_symbol=None*, *group_name=None*, *number=None*)
Bases: `object`

Represents a grouping of several instruments, usually named, and expressed in the score with a group symbol such as a brace or a bracket. In symphonic scores, bracketed part groups usually group families of instruments, such as woodwinds or brass, whereas braces are often used to group multiple instances of the same instrument. See the MusicXML documentation for further information.

Parameters **group_symbol** (*str or None, optional*) – The symbol used for grouping instruments.

group_symbol

Type str or None

name

Type str or None

number

Type int

parent

Type *PartGroup* or None

children

Type list of Part or PartGroup objects

pretty()
Return a pretty representation of this object.

Returns A pretty representation

Return type str

partitura.score.**iter_unfolded_parts**(*part*)
Iterate over unfolded clones of *part*.

For each repeat construct in *part* the iterator produces two clones, one with the repeat included and another without the repeat. That means the number of items returned is two to the power of the number of repeat constructs in the part.

The first item returned by the iterator is the version of the part without any repeated sections, the last item is the version of the part with all repeat constructs expanded.

Parameters **part** (*Part*) – Part to unfold

partitura.score.**unfold_part_maximal**(*part*)
Return the "maximally" unfolded part, that is, a copy of the part where all segments marked with repeat signs are included twice.

Returns **part** – The unfolded Part

Return type *Part*

`partitura.score.`**`make_score_variants`**`(`*part*`)`

Create a list of ScoreVariant objects, each representing a distinct way to unfold the score, based on the repeat structure.

Parameters **`part`** (*`Part`*) – A part for which to make the score variants

Returns List of ScoreVariant objects

Return type list

Notes

This function does not currently support nested repeats, such as in case 45d of the MusicXML Test Suite.

`partitura.score.`**`add_measures`**`(`*part*`)`

Add measures to a part.

This function adds Measure objects to the part according to any time signatures present in the part. Any existing measures will be untouched, and added measures will be delimited by the existing measures.

The Part object will be modified in place.

Parameters **`part`** (*`Part`*) – Part instance

`partitura.score.`**`remove_grace_notes`**`(`*part*`)`

Remove all grace notes from a timeline.

The specified timeline object will be modified in place.

Parameters **`timeline`** (*`Timeline`*) – The timeline from which to remove the grace notes

`partitura.score.`**`expand_grace_notes`**`(`*part*`)`

Expand grace note durations in a part.

The specified part object will be modified in place.

Parameters **`part`** (*`Part`*) – The part on which to expand the grace notes

`partitura.score.`**`iter_parts`**`(`*partlist*`)`

Iterate over all Part instances in partlist, which is a list of either Part or PartGroup instances. PartGroup instances contain one or more parts or further partgroups, and are traversed in a depth-first fashion.

This function is designed to take the result of *`partitura.load_score_midi()`* and *`partitura.load_musicxml()`* as input.

Parameters **`partlist`** (*`list, Part, or PartGroup`*) – A *`partitura.score.Part`* object, *`partitura.score.PartGroup`* or a list of these

Yields *`Part`* instances in *partlist*

`partitura.score.`**`repeats_to_start_end`**`(`*repeats*, *first*, *last*`)`

Return pairs of (start, end) TimePoints corresponding to the start and end times of each Repeat object. If any of the start or end attributes are None, replace it with the end/start of the preceding/succeeding Repeat, respectively, or *first* or *last*.

Parameters

- **`repeats`** (*`list`*) – list of Repeat instances, possibly with None-valued start/end attributes
- **`first`** (`TimePoint`) – The first TimePoint in the timeline
- **`last`** (`TimePoint`) – The last TimePoint in the timeline

Returns list of (start, end) TimePoints corresponding to each Repeat in *repeats*

Return type list

`partitura.score.`**`tie_notes`**(*part*)

Find notes that span measure boundaries and notes with composite durations, and split them adding ties.

Parameters **`part`** (*`Part`*) – Description of *part*

`partitura.score.`**`set_end_times`**(*parts*)

Set missing end times of musical elements in a part to equal the start times of the subsequent element of the same class. This is useful for some classes

Parameters **`part`** (`Part` *`or`* `PartGroup`*`, or list of these`*) – Parts to be processed

`partitura.score.`**`find_tuplets`**(*part*)

Identify tuplets in *part* and set their symbolic durations explicitly.

This function adds *actual_notes* and *normal_notes* keys to the symbolic duration of tuplet notes.

Parameters **`part`** (*`Part`*) – Part instance

`class` `partitura.score.`**`InvalidTimePointException`**

Bases: `object`

Raised when a time point is instantiated with an invalid number.

CHAPTER 5

partitura.performance

This module contains a lightweight ontology to represent a performance in a MIDI-like format. A performance is defined at the highest level by a *PerformedPart*. This object contains performed notes as well as continuous control parameters, such as sustain pedal.

class partitura.performance.**PerformedPart**(*notes*, *id=None*, *part_name=None*, *controls=None*, *sustain_pedal_threshold=64*)

Bases: `object`

Represents a performed part, e.g. all notes and related controller/modifiers of one single instrument.

Performed notes are stored as a list of dictionaries, where each dictionary represents a performed note, should have at least the keys "note_on", "note_off", the onset and offset times of the note in seconds, respectively.

Continuous controls are also stored as a list of dictionaries, where each dictionary represents a control change. Each dictionary should have a key "type" (the name of the control, e.g. "sustain_pedal", "soft_pedal"), "time" (in seconds), and "value" (a number).

Parameters

- **notes** (*list*) – A list of dictionaries containing performed note information.
- **id** (*str*) – The identifier of the part
- **controls** (*list*) – A list of dictionaries containing continuous control information
- **part_name** (*str*) – Name for the part
- **sustain_pedal_threshold** (*int*) – The threshold above which sustain pedal values are considered to be equivalent to on. For values below the threshold the sustain pedal is treated as off. Defaults to 64.

notes

A list of dictionaries containing performed note information.

Type list

id

The identifier of the part

Type str

`part_name`
Name for the part

Type str

`controls`
A list of dictionaries containing continuous control information

Type list

`classmethod from_note_array`(*note_array*, *id=None*, *part_name=None*)
Create an instance of PerformedPart from a note_array. Note that this property does not include non-note information (i.e. controls such as sustain pedal).

`note_array`
Structured array containing performance information. The fields are 'id', 'pitch', 'p_onset', 'p_duration' and 'velocity'.

`sustain_pedal_threshold`
The threshold value (number) above which sustain pedal values are considered to be equivalent to on. For values below the threshold the sustain pedal is treated as off. Defaults to 64.

Based on the control items of type "sustain_pedal", in combination with the value of the "sustain_pedal_threshold" attribute, the note dictionaries will be extended with a key "sound_off". This key represents the time the note will stop sounding. When the sustain pedal is off, *sound_off* will coincide with *note_off*. When the sustain pedal is on, *sound_off* will equal the earliest time the sustain pedal is off after *note_off*. The *sound_off* values of notes will be automatically recomputed each time the *sustain_pedal_threshold* is set.

CHAPTER 6

partitura.musicanalysis

Tools for music analysis.

`partitura.musicanalysis.`**`estimate_voices`**(*notearray*, *monophonic_voices=False*)

Voice estimation using the voice separation algorithm proposed in[1].

Parameters

- **`notearray`** (*`numpy structured array`*) – Structured array containing score information. Required fields are *pitch* (MIDI pitch), *onset* (starting time of the notes) and *duration* (duration of the notes). Additionally, It might be useful to have an *id* field containing the ID's of the notes. If this field is not contained in the array, ID's will be created for the notes.
- **`monophonic_voices`** (*`bool`*) – If True voices are guaranteed to be monophonic. Otherwise notes with the same onset and duration are treated as a chord and assigned to the same voice. Defaults to False.

Returns **voice** – Voice for each note in the notearray. (The voices start with 1, as is the MusicXML convention).

Return type numpy array

References

`partitura.musicanalysis.`**`estimate_key`**(*note_array*, *method='krumhansl'*, **args*, ***kwargs*)

Estimate key of a piece by comparing the pitch statistics of the note array to key profiles[2],[3].

Parameters

- **`note_array`** (*`structured array`*) – Array containing the score

[1] Chew, E. and Wu, Xiaodan (2004) "Separating Voices in Polyphonic Music: A Contig Mapping Approach". In Uffe Kock, editor, "Computer Music Modeling and Retrieval". Springer Berlin Heidelberg.

[2] Krumhansl, Carol L. (1990) "Cognitive foundations of musical pitch", Oxford University Press, New York.

[3] Temperley, D. (1999) "What's key for key? The Krumhansl-Schmuckler key-finding algorithm reconsidered". Music Perception. 17(1), pp. 65–100.

- **method** (*{'krumhansl', 'temperley'}*) – Method for estimating the key. Default is 'krumhansl'.
- **kwargs** (*args,*) – Positional and Keyword arguments for the key estimation method

Returns

- **root** (*str*) – Root of the key (key name)
- **mode** (*str*) – Mode of the key ('major' or 'minor')
- **fifths** (*int*) – Position in the circle of fifths

References

partitura.musicanalysis.**estimate_spelling**(*note_array*, *method='ps13s1'*, **args*, ***kwargs*)

Estimate pitch spelling using the ps13 algorithm[4],[5].

Parameters

- **note_array** (*structured array*) – Array with score information
- **method** (*str (default 'ps13s1')*) – Pitch spelling algorithm. More methods will be added.
- ***args** – positional arguments for the algorithm specified in *method*.
- ****kwargs** – Keyword arguments for the algorithm specified in *method*.

Returns **spelling** – Array with pitch spellings. The fields are 'step', 'alter' and 'octave'

Return type structured array

References

[4] Meredith, D. (2006). "The ps13 Pitch Spelling Algorithm". Journal of New Music Research, 35(2):121.

[5] Meredith, D. (2019). "RecurSIA-RRT: Recursive translatable point-set pattern discovery with removal of redundant translators". 12th International Workshop on Machine Learning and Music. Würzburg, Germany.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index