Musictheorpy

Jan 25, 2020

Contents

CHAPTER 1

License

1.1 Questions?

If you have any questions about the Musictheorpy terms of use, please send an email to jeff.moorhead1@gmail.com.

CHAPTER 2

Installation

Installing Musictheorpy is really easy! All you need to do is run `pip install musictheorpy` and you can begin using the library right away. Or, if you do not want to use pip, you can also download the source code from Github and install the code directly from within the project root directory.

CHAPTER 3

Musictheorpy API Reference

Musictheorpy is a Python library to perform musical calculations, including intervals, triads/chords, and scales.

3.1 Quickstart

All of the classes below can be accessed directly from within the musictheorpy namespace, for example:

```
>>> import musictheorpy
>>> note = musictheorpy.Note('A')
```

3.1.1 Notes

class Note(*qualified_name*)

Note objects are composed of a string representing a qualified note name. The qualified note name should be a letter A through G, optionally followed by a qualifier. Valid qualifiers are *#*, *##*, *b*, and *bb* (lowercase B). A note name with no qualifier is also allowed and represents a natural. For example, *Note('A')*, *Note('C#')*, and *Note('Gb')* are all valid, while *Note('H')* and *Note('Fbbb')* are invalid. Note that note names can be lowercase or uppercase, but qualifiers must be lowercase (e.g. *Note('ab')* is valid, but *Note('aB')* is not). If you attempt to create a Note object with an invalid qualified note name, a NoteNameError is raised.

3.1.2 Scales

class Scale(*qualified_name*)

A scale object is constructed with a string representing the qualified scale name. The qualified scale name consists of a tonic followed by a scale quality. Valid tonics are letters A through G. Valid qualities are major, harmonic minor, melodic minor, and natural minor. If an invalid tonic passed, an InvalidTonicError is raised. If an invalid quality is passed, an InvalidQualityError is raised. For example,

```
>>> c = Scale('C major')
>>> c_harm = Scale('C harmonic minor')
>>> c_badquality = Scale('C Foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/scales.py", line 36, in __
↪init__
    super().__init__('SCALE', qualified_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/notegroups.py", line 74,␣
↪in __init__
    raise InvalidQualityError("Quality %s is not valid" % self.quality) from None
musictheorpy.notegroups.InvalidQualityError: Quality Foo is not valid
>>>
>>> c_badtonic = Scale('Z major')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/scales.py", line 36, in __
↪init__
    super().__init__('SCALE', qualified_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/notegroups.py", line 68,␣
↪in __init__
    self._validate_root(unpacked_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/scales.py", line 66, in _
↪validate_root
    "key signature." % unpacked_name['ROOT'])
musictheorpy.scales.InvalidTonicError: Invalid tonic: Z. It is possible that this␣
↪tonic is a valid
note name but that building the desired scale from this note would result in a scale␣
↪with an invalid
key signature.
```

As indicated in the trace back for `c_badtonic` above, it is possible to pass a valid note name, but still receive an InvalidTonicError. This occurs when the key signature of the given scale name would include qualifiers beyond sharps and flats. For example, G# Major would have F## in its key signature. Because key signatures like this are generally not used in music theory, they are not valid.

The key signature of a Scale object is accessible through its key_signature property, which is a tuple of strings representing the notes that make up the scale's key signature. For example,

```
>>> a_major = Scale('A major')
>>> a_major.key_signature
('F#', 'C#', 'G#')
>>> e_minor = Scale('E natural minor')
>>> 'F#' in e_minor.key_signature
True
```

In addition, you can access all the notes in the scale through the object's `notes` attribute, which provides a tuple of strings representing all the notes in the scale.

Scale objects implement the `__getitem__` and `__contains__` magic methods. `__getitem__` allows you to lookup notes in a scale by degree name. Valid degree names are tonic, supertonic, mediant, subdominant, dominant, submediant, and leading tone. For example,

```
>>> a_major = Scale('A major')
>>> a_major['tonic']
'A'
>>> a_major['submediant']
'F#'
```

Finally, users can test if a note is in a given scale using Python's built-in `in` keyword, thanks to the `__contains__` method.

```
>>> a_major = Scale('A major')
>>> 'F#' in a_major
True
>>> 'B#' in a_major
False
```

3.1.3 Chords

class Chord(*qualified_name*)

Chord objects are constructed with a string representing the qualified name of the chord. Like scales, the qualified name of a chord is made up of a bass note name (letters A through G) followed by a quality. Valid chord qualities are major, minor, diminished, augmented, and minor 7b5. Chords containing upper extensions 7, 9, 11, and 13 are also possible. All upper extensions can be dominant, major, or minor, e.g. dominant 7, major 9, minor 13. In addition, extensions 9, 11, and 13 can be modified with a flat (b) or sharp (#) for dominant chords, e.g. dominant #9, dominant b13.

If an invalid bass note is specified, an InvalidBassError is raised. Similarly, if an invalid chord quality is specified, an InvalidQualityError is raised. For example,

```
>>> c = Chord('C major')
>>> c_seventh = Chord('C dominant 7')
>>> z = Chord('Z major')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/chords.py", line 10, in __
↪init__
    super().__init__('CHORD', qualified_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/notegroups.py", line 68,␣
↪in __init__
    self._validate_root(unpacked_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/chords.py", line 25, in _
↪validate_root
    raise InvalidBassError("Invalid bass note: %s" % unpacked_name['ROOT'])
musictheorpy.chords.InvalidBassError: Invalid bass note: Z
>>>
>>> c_badqual = Chord('C FOO')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/chords.py", line 10, in __
↪init__
    super().__init__('CHORD', qualified_name)
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/notegroups.py", line 74,␣
↪in __init__
    raise InvalidQualityError("Quality %s is not valid" % self.quality) from None
musictheorpy.notegroups.InvalidQualityError: Quality FOO is not valid
```

Users can access the notes in a Chord object via the object's `notes` attribute. This attribute provides a tuple containing all the notes in the chord as strings. For example,

```
>>> c_dominant = Chord('C dominant 7')
>>> c_dominant.notes
('C', 'E', 'G', 'Bb')
```

In addition, Chord objects implement the __contains__ method so users can check if a note is in the chord directly:

```
>>> c = Chord('C major')
>>> 'E' in c
True
>>> 'F' in c
False
```

Finally, Chord objects allow access to its constituent notes via the __getitem__ method, which allows lookup by degree name. Valid degree names are bass, third, fifth, seventh, ninth, eleventh, and thirteenth. Note that not all degrees apply to all chords, and only thirteenth chords will have all degrees. In general, chords only contain a subset of these degrees. If the caller tries to access a degree that is not present in the given chord, __getitem__ returns None. For example,

```
>>> c = Chord('C major')  # a triad, no extensions
>>> c['third']  # valid degree
'E'
>>> c['ninth'] is None  # C triad does not have a ninth
True
```

If an invalid degree is passed, an InvalidDegreeError is raised:

```
>>> c = Chord('C major')
>>> c['foo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jmoorhead/projects/musictheorpy/musictheorpy/chords.py", line 18, in __
↪getitem__
    raise InvalidDegreeError("Invalid degree name: %s" % element) from None
musictheorpy.notegroups.InvalidDegreeError: Invalid degree name: foo
```

3.2 Reference

3.2.1 Notes

VALID_NOTES
Valid note names, not including qualifiers.

VALID_QUALIFIERS
Valid note qualifiers.

VALID_QUALIFIED_NAMES
Valid note names, including qualifiers.

class musictheorpy.notes.**Note**(*qualified_name*)
Represents an individual note, the atomic element of Western music theory. The note's qualified name is the note name, optionally followed by the qualifier. Valid note names are English capital letters A through G. Valid qualifiers are # (sharp), b (flat), ## (double sharp), bb (double flat). Note that flats must be lowercase or a NoteNameError is raised. A note name followed by no qualifier represents a natural, e.g. 'A' represents 'A natural'. If an invalid qualified note name is passed, a NoteNameError is raised. Once a Note object is created, it's qualified name should not be modified.

ascend_interval(*qualified_interval_name*)

Parameters **qualified_interval_name** – The interval to ascend, e.g. major 3.

Returns A Note object representing the top note of the interval. If the top note of the interval is not a valid note, such as F###, and InvalidIntervalError is raised.

descend_interval(*qualified_interval_name*)

Parameters **qualified_interval_name** – the interval to fetch, e.g. major 3.

Returns a Note object representing the bottom note of the interval. If the bottom note of the interval is not a valid note, such as F###, an InvalidInvervalError is raised.

get_interval_name(*top_note*)
Gives the name of the interval between the current note and `top_note`. Note that the current note is always treated as the bottom note in the interval.

Parameters **top_note** (*str*) – the qualified name of the top note of the interval.

exception musictheorpy.notes.**NoteNameError**
Raised when attempting to create a Note object with a qualified note name that is not valid.

3.2.2 Scales

class musictheorpy.scales.**Scale**(*qualified_name*)
Represents a collection of notes. Scales are built from a series of whole and half steps and have a key signature and tonic. Each note in a scale is identified either by a number (1 through 7) or a degree name. Valid tonics are English letters A through G, and valid qualities are MAJOR, HARMONIC MINOR, MELODIC MINOR, and NATURAL MINOR. An InvalidTonicError is raised if the scale name has a key signature involving double sharps or double flats.

classmethod _fetch_key_signature(*tonic*, *quality*)
Returns a list of strings with the note names that make up the scale's key signature.

_validate_root(*unpacked_name*)
Ensures that the tonic of the scale is valid.

Parameters **unpacked_name** (*list*) – The qualified name of the scale unpacked into a list.

Raises InvalidTonicError: If the tonic is an invalid note or if the key signature of the scale would contain double sharps or double flats.

get_parallel()
Returns the parallel major or minor of the current scale.

Return type *Scale*

get_relative()
Returns the relative major or minor of the current scale.

Return type *Scale*

get_triad_for_degree(*degree*)
Builds a triad based on the current scale's quality and the given degree.

Parameters **degree** – a string representing the scale degree, such as TONIC, MEDIANT.

Raises InvalidDegreeError

Returns a Chord object with the scale degree as its root.

exception musictheorpy.scales.**InvalidTonicError**
Raised when attempting to create a Scale object with an invalid tonic. This situation could arise from attempting to create a Scale object with a tonic that is greater than G, or when attempting to create a Scale object with a valid tonic name (that is an English letter between A and G), but for whom the passed quality would result in a

scale that includes invalid note names in its key signature, such as double sharps. An example of this situation is a G# Major scale.

3.2.3 Chords

class musictheorpy.chords.**Chord**(*qualified_name*)
Represents triads and chords. Triads have a quality of either major or minor, and consist of a bass note, a third, and a fifth. Chords are triads with upper extensions. Possible upper extensions are 7, 9, 11, and 13, optionally qualified with a sharp (#) or flat (b).

_validate_root(*unpacked_name*)
Ensures that the bass of the chord is valid.

Parameters **unpacked_name** (*dict*) – The qualified name of the chord unpacked into a list.

Raises InvalidBassError: If the bass note is an invalid note.

exception musictheorpy.chords.**InvalidBassError**
Raised when attempting to create a Chord object with an invalid bass note.

3.2.4 Note Groups

class musictheorpy.notegroups.**_NoteGroup**(*grouptype*, *qualified_name*)
An abstract base class used to define an interface for note groups, such as scales and chords.

exception musictheorpy.notegroups.**InvalidDegreeError**
Raised when an attempting to fetch an invalid scale degree name. Valid scale degree names are tonic, supertonic, mediant, subdominant, dominant, submediant, and leading tone.

exception musictheorpy.notegroups.**InvalidQualityError**
Raised when the quality of a scale or chord is invalid.

3.2.5 Interval Utils

musictheorpy.interval_utils.**INTERVAL_NOTE_PAIRS**
A dictionary giving note names corresponding to intervals. The keys in the dictionary are starting notes. Each dictionary element has a dictionary as its value. The keys of the inner dictionaries are the interval code corresponding to a qualified interval name. The values of the inner dictionaries are the top note of the interval.

class musictheorpy.interval_utils.**_IntervalBuilder**(*rootnote*)
This class can be used as a utility class to facilitate interval calculations. Objects store a root note as a string and builds intervals on that note.

ascend_interval_from_name(*qualified_interval_name*)
Ascend a specified interval from the root note.

Raises InvalidIntervalError: If ascending the specified interval from the root would result in an invalid note, such as a triple sharp or triple flat.

descend_interval_from_name(*qualified_interval_name*)
Descend a specified interval from the root note.

Raises InvalidIntervalError: If descending the specified interval from the root would result in an invalid note, such as a triple sharp or triple flat.

get_interval_name(*root_note*, *top_note*)

Parameters

- **root_note** (*str*) – the bottom note of the interval.
- **top_note** (*str*) – the top note of the interval.

Returns the name of the interval between the bottom and top notes.

exception musictheorpy.interval_utils.**InvalidIntervalError**

Raised when an interval would result in an invalid top (ascending) or bottom (descending) note, e.g. a diminished third ascending from Gb would technically be a Bbbb (B triple flat). While this is enharmonically equivalent to Ab, Gb to Ab is a major second, not a diminished third. Because the technically correct note is not a valid note name, an InvalidIntervalError should be raised.

CHAPTER 4

Contributing

Musictheorpy is under active development and will grow to include new features in the future. If you are interested in contributing to the project, all you need to do is fork the Github repository, download the source code, and begin working. Continue to the complete guide below for more details. Even if you don't want to contribute directly, I am always open to hearing people's ideas, so if you have a thought on how the project can improve, please feel free to send me an email at jeff.moorhead1@gmail.com with your name, thoughts on improving the library, and how I can best contact you to continue the discussion. I only ask that you be patient, as between work, graduate school, and life, it may take me a few days to respond.

4.1 Setting Up Your Environment

The first step to contributing code to Musictheorpy is to fork the repository on Github and download a copy to your computer so you can make your changes. See git-scm for more information on contributing to projects on Github.

You can run the unit tests using either Python's built in `unittest` module, or by installing `Nose`, and running `nosetests` from the project root:

```
# from the project root directory
>>> python -m unittest discover

# or, using Nose...
>>> nosetests
```

If you do not have Nose, you can create a virtual environment and install Nose with `pip install nose`. See the Python docs for more information about virtual environments.

All changes are required to have accompanying unit tests. Untested changes will be rejected. Further, any new classes, methods, or functions are required to contain a docstring describing the new functionality and any parameters present in the signature.

Once you have finished your changes and have a passing test suite, please submit a pull request to have your changes merged. More information on pull requests can be found here.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

G

I

M

N

S

V