chordparser

Release 0.4.2

Mar 26, 2023

Contents

1 Contents **3**
- 1.1 Introduction . . . 3
- 1.2 Installation . . . 4
- 1.3 Tutorial . . . 4
- 1.4 How-to Guides . . . 4
- 1.5 Reference . . . 4
- 1.6 Background . . . 28
- 1.7 Credits . . . 28
- 1.8 Contributing . . . 28
- 1.9 Licence . . . 30
- 1.10 History . . . 30

2 Indices and tables **33**

Index **35**

chordparser is a Python 3 package that provides a musical framework for analysing chords. The *Introduction* covers what it is and why it is.

If you are a musician, not a programmer, and you just want to see chordparser in action, the Colab notebook is for you. It showcases the functions of chordparser and you can even use it for your own chord sheets without knowing any programming!

If you are a programmer however, then get started with *Installation* and jump right into using chordparser with the *hands-on Tutorial*. The *How-to guides* provide step-by-step guides on using different aspects of chordparser, while the *Reference* gives a full technical reference to all of chordparser's classes. Finally, the *Background* section provides explanation and discussion of key topics.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 What is chordparser?

chordparser aims to provide a framework for harmonic analysis of chords. This means that chordparser can help in understanding how each chord functions and use roman numeral chord notation to describe them.

chordparser comes with the Parser, a one-stop shop for creating, manipulating and analysing musical objects. The Parser can take in chord notation to create a Chord, and analyse it against other chords or keys.

To do this, chordparser has several musical classes that serve as the foundation for analysing chords. These include Notes, Keys, Scales, Chords and Romans. Chords and Romans have been built on the other classes so they can be analysed easily.

1.1.2 Why chordparser?

Current songbooks and chord sheet applications can display chord sheets beautifully and even transpose the chords (though wrongly sometimes). However, these only stop there and cannot aid in analysing chord progression despite having all the chords. chordparser seeks to take a step further by allowing musicians to analyse chords in a chord sheet while reading them and deepen their understanding of the song's chord structure.

With chordparser's musical framework, chord structures from a basic V-I cadence to more complicated secondary chords can be analysed. Here are some examples of chords that can be analysed for: diatonic chords, mode mixture/ borrowed chords, secondary dominant chords (e.g. V/V), secondary leading tone chords.

Possible applications of chordparser include incorporating it into a songbook application or reading and analysing your own chord sheets. Check out the *Tutorial* on how you can do that and more.

1.2 Installation

1.2.1 Stable release

To install chordparser, run this command in your terminal:

```
$ pip install chordparser
```

This is the preferred method to install chordparser, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2.2 From sources

The sources for chordparser can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/titus-ong/chordparser
```

Or download the tarball:

```
$ curl -OJL https://github.com/titus-ong/chordparser/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.3 Tutorial

chordparser's tutorial can be found at the Colab notebook. Simply follow along the code cells, and you'll learn how to write a script that can read a chord sheet, transpose it, convert the chords to roman numeral notation and analyse them.

1.4 How-to Guides

Work in progress.

1.5 Reference

Important: Many examples will use individual *Editors* and *Analysers* to create and interact with musical objects. Do have a look at the *Parser* and see how it makes everything much easier. *You should NOT need to initialise any of the individual Editors or Analysers!*

1.5.1 Parser (All-in-One Editor and Analyser)

class chordparser.**Parser**

A class that acts as a central collection for *Editors* and *Analysers*.

The *Parser* inherits all the various *Editors* and *Analysers*. As such, all the examples using the *Editors* and *Analysers* can also use the *Parser* to create and interact with musical objects. This makes it more convenient to initialise the various musical classes without having to initialise many different *Editors* for each class beforehand.

Examples

```
>>> cp = Parser()
>>> cp.create_note("D#")
D note
>>> c_chord = cp.create_chord("Cmaj7/G")
>>> c_chord
Cmaj7/G chord
>>> c_scale = cp.create_scale("C", "major")
>>> cp.analyse_diatonic(c_chord, c_scale)
[(IM43 roman chord, 'major', None)]
```

analyse_all(*chord*, *scale*, *incl_submodes=False*, *allow_power_sus=False*, *default_power_sus='M'*)

Analyse if a *Chord* is diatonic to a *Scale* for any mode.

The *Chord* is analysed against the *Scale* as well as the other modes of the *Scale*.

Parameters

- **chord** (Chord) – The *Chord* to be analysed.
- **scale** (Scale) – The *Scale* to check against.
- **incl_submodes** (*boolean, Optional*) – Selector to include the minor submodes if *scale* is minor. Default False when optional.
- **allow_power_sus** (*boolean, Optional*) – Selector to allow power and sus chords when analysing them. Default False when optional.
- **default_power_sus** (*{"M", "m"}, Optional*) – The default quality to convert power and sus chords to if analysing them. "M" is major and "m" is minor.

Returns A list of information on the *Chord* if it is diatonic. The first *str* is the *mode* of the scale it is diatonic to and the second *str* is the *submode*. The list is empty if the *Chord* is not diatonic.

Return type list of (*Roman*, str, str)

Examples

```
>>> CE = ChordEditor()
>>> SE = ScaleEditor()
>>> CA = ChordAnalyser()
```

Diatonic chords

```
>>> c_scale = SE.create_scale("C", "minor")
>>> degree_1 = CE.create_diatonic(c_scale, 1)
>>> CA.analyse_all(degree_1, c_scale)
[(i roman chord, 'minor', 'natural'), (i roman chord, 'dorian', None), (i␣
↪roman chord, 'phrygian', None)]
```

Checking against minor submodes

```
>>> degree_7 = CE.create_chord("G")
>>> CA.analyse_diatonic(degree_7, c_scale)
[(V roman chord, 'major', None), (V roman chord, 'lydian', None)]
>>> CA.analyse_diatonic(degree_7, c_scale, incl_submodes=True)
[(V roman chord, 'minor', 'harmonic'), (V roman chord, 'minor', 'melodic'),␣
↪(V roman chord, 'major', None), (V roman chord, 'lydian', None)]
```

Analysing power/sus chords

```
>>> power = CE.create_chord("C5")
>>> CA.analyse_diatonic(power, c_scale)
[]
>>> CA.analyse_diatonic(power, c_scale, allow_power_sus=True, default_power_
↪sus="m")
[(I roman chord, 'major', None), (I roman chord, 'mixolydian', None), (I␣
↪roman chord, 'lydian', None)]
```

analyse_diatonic(*chord*, *scale*, *incl_submodes=False*, *allow_power_sus=False*, *default_power_sus='M'*)

Analyse if a *Chord* is diatonic to a *Scale*.

There may be multiple tuples in the returned list if submodes are included.

Parameters

- **chord** (Chord) – The *Chord* to be analysed.
- **scale** (Scale) – The *Scale* to check against.
- **incl_submodes** (*boolean, Optional*) – Selector to include the minor submodes if *scale* is minor. Default False when optional.
- **allow_power_sus** (*boolean, Optional*) – Selector to allow power and sus chords when analysing them. Default False when optional.
- **default_power_sus** (*{"M", "m"}, Optional*) – The default quality to convert power and sus chords to if analysing them. "M" is major and "m" is minor.

Returns A list of information on the *Chord* if it is diatonic. The first *str* is the *Scale*'s *mode* and the second *str* is the *Scale*'s *submode*. The list is empty if the *Chord* is not diatonic.

Return type list of (*Roman*, str, str)

Examples

```
>>> CE = ChordEditor()
>>> SE = ScaleEditor()
>>> CA = ChordAnalyser()
```

Diatonic chords

```
>>> c_scale = SE.create_scale("C", "minor")
>>> degree_1 = CE.create_diatonic(c_scale, 1)
>>> CA.analyse_diatonic(degree_1, c_scale)
[(i roman chord, 'minor', 'natural')]
```

Checking against minor submodes

```
>>> degree_7 = CE.create_chord("G")
>>> CA.analyse_diatonic(degree_7, c_scale)
[]
>>> CA.analyse_diatonic(degree_7, c_scale, incl_submodes=True)
[(VII roman chord, 'minor', 'harmonic'), (VII roman chord, 'minor', 'melodic
↪')]
```

Analysing power/sus chords

```
>>> power = CE.create_chord("C5")
>>> CA.analyse_diatonic(power, c_scale)
[]
>>> CA.analyse_diatonic(power, c_scale, allow_power_sus=True, default_power_
↪sus="m")
[(i roman chord, 'minor', 'natural')]
```

analyse_secondary(*prev_chord*, *next_chord*, *scale*, *incl_submodes=False*, *allow_power_sus=False*, *default_power_sus='M'*, *limit=True*)

Analyse if a *Chord* has a secondary function.

Check if a *Chord* is a secondary chord. By default, only secondary dominant and secondary leading tone chords are checked for.

Parameters

- **prev_chord** (Chord) – The *Chord* to be analysed for secondary function.
- **next_chord** (Chord) – The *Chord* to be tonicised.
- **scale** (Scale) – The *Scale* to check against.
- **incl_submodes** (*boolean, Optional*) – Selector to include the minor submodes if *scale* is minor. Default False when optional.
- **allow_power_sus** (*boolean, Optional*) – Selector to allow power and sus chords when analysing them. Default False when optional.
- **default_power_sus** (*{"M", "m"}, Optional*) – The default quality to convert power and sus chords to if analysing them. "M" is major and "m" is minor.
- **limit** (*boolean, Optional*) – Selector to only check for secondary dominant and leading tone chords. Default True when optional.

Returns The secondary chord notation `prev_roman/next_roman`.

Return type str

Examples

```
>>> CE = ChordEditor()
>>> SE = ScaleEditor()
>>> CA = ChordAnalyser()
```

Diatonic chords

```
>>> c_scale = SE.create_scale("C")
>>> g = CE.create_diatonic(c_scale, 5)
>>> d = CE.create_chord("D")
>>> CA.analyse_secondary(d, g, c_scale)
'V/V'
```

Checking against minor submodes

```
>>> vii = CE.create_chord("C#dim7")
>>> degree_2 = CE.create_diatonic(c_scale, 2)
>>> CA.analyse_secondary(vii, degree_2, c_scale)
''
>>> CA.analyse_secondary(vii, degree_2, c_scale, incl_submodes=True)
'vii°7/ii'
```

Analysing power/sus chords

```
>>> power = CE.create_chord("D5")
>>> g = CE.create_diatonic(c_scale, 5)
>>> CA.analyse_secondary(power, g, c_scale)
''
>>> CA.analyse_secondary(power, g, c_scale, allow_power_sus=True)
'V/V'
```

Analysing other secondary chords

```
>>> e = CE.create_chord("Em")
>>> CA.analyse_secondary(e, g, c_scale)
''
>>> CA.analyse_secondary(e, g, c_scale, limit=False)
'vi/V'
```

change_chord(*chord*, *root=None*, *quality=None*, *add=None*, *remove=None*, *bass=None*, *inplace=True*)

Change a *Chord*'s attributes.

Alter parts of the *Chord* by specifying the standard chord notation for each attribute.

Parameters

- **chord** (Chord) – The *Chord* to be changed.
- **root** (*str, Optional*) – The root of the *Chord*.
- **quality** (*str, Optional*) – The quality of the *Chord*.
- **add** (*str, Optional*) – The added notes of the *Chord*.
- **remove** (*str, Optional*) – The added notes of the *Chord* that are to be removed.
- **bass** (*str or boolean, Optional*) – The bass note of the *Chord*. Specify False to remove the bass note.
- **inplace** (*boolean, Optional*) – Selector to change the current *Chord* or to return a new *Chord*. Default True when optional.

Returns The changed *Chord*.

Return type *Chord*

Examples

```
>>> CE = ChordEditor()
>>> c = CE.create_chord("C7add4/G")
>>> CE.change_chord(c, root="D#", quality="maj7", add="b6", remove="4",␣
↪bass=False)
```

(continues on next page)

(continued from previous page)

```
Dmaj76 chord
>>> CE.change_chord(c, root="E", quality="m", bass="C#", inplace=False)
Em6/C chord
>>> c
Dmaj76 chord
```

change_key(*key*, *root=None*, *mode=None*, *submode=None*, *inplace=True*)

Change a *Key*'s *root*, *mode* and/or *submode* attributes.

The root note must be a valid *Note* notation if type *str*. The submode refers to the different types of 'minor'/ 'aeolian' mode, i.e. 'natural', 'harmonic' and 'melodic'. Hence, other than the 'minor'/ 'aeolian' mode, the submode must be None.

Parameters

- **key** (Key) – The *Key* which attributes you want to change.
- **root** (*Note, Optional*) – The *root* of the *Key* to be changed.
- **mode** (*{None, 'major', 'minor', 'ionian', 'dorian', 'phrygian', 'lydian', 'mixolydian', 'aeolian', 'locrian'}, Optional*) – The *mode* of the *Key* to be changed.
- **submode** (*{None, 'natural', 'harmonic', 'melodic'}, Optional*) – The submode of the *Key* to be changed. Default 'natural' for the 'minor'/ 'aeolian' mode and None for the other modes when optional.
- **inplace** (*boolean, optional*) – Selector to change the notation of current *Key* or to return a new *Key*. Default True when optional.

Returns The *Key* with the new attributes.

Return type *Key*

Examples

```
>>> KE = KeyEditor()
>>> c = KE.create_key("C", "dorian")
>>> KE.change_key(c, root="D", mode="minor", submode="harmonic")
D harmonic minor
>>> KE.change_key(c, mode="major", inplace=False)
D major
>>> c
D harmonic minor
```

change_note(*note*, *notation*, *inplace=True*)

Change a *Note*'s notation.

Accepts a-g or A-G and optional accidental symbols (b, bb, #, ##, or their respective unicode characters , , , or).

Parameters

- **note** (Note) – The *Note* which value you want to change.
- **notation** (*str*) – The new notation for the *Note*.
- **inplace** (*boolean, optional*) – Selector to change the notation of current *Note* or to return a new *Note*. Default True when optional.

Returns The *Note* with the new notation.

Return type *Note*

Examples

```
>>> NE = NoteEditor()
>>> a = NE.create_note("A")
>>> NE.change_note(a, "Bb")
B note
>>> NE.change_note(a, "C#", inplace=False)
C note
>>> a
B note
```

change_scale(*scale*, **args*, *inplace=True*, ***kwargs*)

Change a *Scale* based on its *Key*.

The parameters accepted are the same parameters accepted for changing a *Key*: the *root*, *mode* and *submode*. The *root* must be a valid *Note* notation if type *str*. The *submode* refers to the different types of 'minor'/ 'aeolian' mode, i.e. 'natural', 'harmonic' and 'melodic'. Hence, other than the 'minor'/ 'aeolian' *mode*, the *submode* must be None.

Parameters

- **scale** (*Scale*) – The *Scale* which key you want to change.
- ***args** (*iterable*) – The parameters for changing the *Scale*'s *Key*.
- **inplace** (*boolean, Optional*) – Selector to change the current *Scale* or to return a new *Scale*. Default True when optional.
- ****kwargs** (*dict*) – The keyword parameters for changing the *Scale*'s *Key*.

Returns The *Scale* with the new *Key*.

Return type *Scale*

Examples

```
>>> SE = ScaleEditor()
>>> c = SE.create_scale("C", "major")
>>> SE.change_scale(c, "D", "lydian")
D lydian scale
>>> SE.change_scale(c, "E", "dorian", inplace=False)
E dorian scale
>>> c
D lydian scale
```

create_chord(*notation*)

Create a *Chord*.

Parameters **notation** (*str*) – The *Chord* notation. Standard chord notation[1] is accepted.

Returns The created *Chord*.

Return type *Chord*

[1] 'Chord letters' (2020) *Wikipedia*. Available at https://en.wikipedia.org/wiki/Chord_letters (Accessed: 28 July 2020)

Raises

- `SyntaxError` – If the notation is invalid.
- `SyntaxError` – If the string of added notes is invalid.

References

Examples

```
>>> CE = ChordEditor()
>>> CE.create_chord("C#dim7addb4/E")
Cdim7add4/E chord
>>> CE.create_chord("Ebsus4add#9/G")
Esus9/G chord
```

create_diatonic(*scale_key*, *degree=1*)

Create a diatonic *Chord* from a *Scale* or *Key*.

Parameters

- **scale_key** (*Scale or Key*) – The *Scale* or *Key* to create the diatonic *Chord* from.
- **degree** (*int, Optional*) – The scale degree of the diatonic *Chord*. Default 1 when optional.

Returns The created diatonic *Chord*.

Return type *Chord*

Raises `ValueError` – If *degree* is not in the range [1, 7].

Examples

```
>>> KE = KeyEditor()
>>> SE = ScaleEditor()
>>> c_key = KE.create_key("C", "major")
>>> c_scale = SE.create_scale(c_key)
>>> CE = ChordEditor()
>>> CE.create_diatonic(c_key, 3)
Em chord
>>> CE.create_diatonic(c_scale, 7)
Bdim chord
```

create_key(*root*, *mode='major'*, *submode=None*)

Create a *Key* from a root note, mode and submode.

The root note must be a valid *Note* notation if type *str*. The submode refers to the different types of minor/aeolian mode, i.e. natural, harmonic and melodic. Hence, other than the 'minor'/ 'aeolian' mode, the submode must be None.

Parameters

- **root** (*Note or str*) – The root note of the *Key*.
- **mode** (*{'major', 'minor', 'ionian', 'dorian', 'phrygian', 'lydian', 'mixolydian', 'aeolian', 'locrian'}, Optional*) – The mode of the *Key*.

- **submode** (*{None, 'natural', 'harmonic', 'melodic'}, Optional*) – The submode of the *Key*. Default 'natural' for the 'minor'/ 'aeolian' mode and None for the other modes when optional.

Returns The created *Key*.

Return type *Key*

Raises

- *ModeError* – If the *mode* is not 'minor'/ 'aeolian' and the *submode* has been specified.
- SyntaxError – If the *mode* is 'minor'/ 'aeolian' and the *submode* is invalid.

Examples

```
>>> KE = KeyEditor()
>>> KE.create_key("C")
C major
>>> KE.create_key("D", "dorian")
D dorian
>>> KE.create_key("E", "minor")
E natural minor
>>> KE.create_key("F", "minor", "harmonic")
F harmonic minor
```

create_note(*notation*)

Create a *Note* from its notation.

Accepts a-g or A-G and optional accidental symbols (b, bb, #, ##, or their respective unicode characters , , , or).

Parameters **notation** (*str*) – The notation of the *Note*.

Returns A *Note* object with value equal to its notation.

Return type *Note*

Raises SyntaxError – If the notation does not follow accepted notation.

Examples

```
>>> NE = NoteEditor()
>>> NE.create_note("C")
C note
>>> NE.create_note("D#")
D note
```

create_scale(*value*, **args*, ***kwargs*)

Create a *Scale*.

Specify either a *Key* or the parameters necessary to create a *Key*.

Parameters

- **value** – Either a *Key* or the first parameter for creating a *Key* (i.e. the *root*).
- ***args** (*iterable*) – The parameters for creating a *Key*.
- ****kwargs** (*dict*) – The keyword parameters for creating a *Key*.

Returns The created *Scale*.

Return type *Scale*

See also:

chordparser.KeyEditor.create_key() See the necessary parameters for creating a *Key*.

Examples

```
>>> SE = ScaleEditor()
>>> KE = KeyEditor()
>>> c_key = KE.create_key("C", "minor")
>>> SE.create_scale(c_key)
C natural minor scale
>>> SE.create_scale("C", "major")
C major scale
```

get_intervals(**notes*)

Get the semitone intervals between *Notes*.

Multiple *Notes* as arguments are accepted. The interval for each *Note* is relative to the previous *Note*.

Parameters ***notes** (Note) – Any number of *Notes*.

Returns The tuple of semitone intervals between all adjacent *Notes*.

Return type tuple of int

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> f = NE.create_note("F")
>>> a = NE.create_note("A")
>>> NE.get_intervals(c, f, a)
(5, 4)
```

get_letter_intervals(**notes*)

Get the letter intervals between *Notes*.

Multiple *Notes* as arguments are accepted. The interval for each *Note* is relative to the previous *Note*.

Parameters ***notes** (Note) – Any number of *Notes*.

Returns The tuple of letter intervals between all adjacent *Notes*.

Return type tuple of int

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> f = NE.create_note("F")
>>> a = NE.create_note("A")
>>> NE.get_letter_intervals(c, f, a)
(3, 2)
```

get_min_intervals(**notes*)

Get the shortest semitone distance between *Notes*.

Multiple *Notes* as arguments are accepted. The distance for each *Note* is relative to the previous *Note*.

Parameters ***notes** (Note) – Any number of *Notes*.

Returns The tuple of the shortest semitone distances between all adjacent *Notes*.

Return type tuple of int

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> b = NE.create_note("B")
>>> NE.get_intervals(c, b)
(11,)
>>> NE.get_min_intervals(c, b)
(-1,)
```

get_tone_letter(**notes*)

Get the semitone and letter intervals between *Notes*.

Multiple *Notes* as arguments are accepted. The intervals for each *Note* are relative to the previous *Note*.

Parameters ***notes** (Note) – Any number of *Notes*.

Returns The nested tuple of semitone and letter intervals between all adjacent *Notes*. The inner tuple is the semitone and letter intervals between two adjacent *Notes*.

Return type tuple of (int, int)

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> f = NE.create_note("F")
>>> a = NE.create_note("A")
>>> NE.get_tone_letter(c, f, a)
((5, 3), (4, 2))
```

relative_major(*key*)

Change a *Key* to its relative major.

The *Key*'s *mode* must be 'minor'/ 'aeolian'.

Parameters **key** (Key) – The *Key* to be changed.

Raises *ModeError* – If the *Key* is not 'minor'/ 'aeolian'.

Examples

```
>>> KE = KeyEditor()
>>> key = KE.create_key("D", "minor")
>>> KE.relative_major(key)
D major
```

relative_minor(*key*, *submode='natural'*)
Change a *Key* to its relative minor.

The *Key*'s *mode* must be 'major'/ 'ionian'.

Parameters

- **key** (Key) – The *Key* to be changed.
- **submode** (*{'natural', 'harmonic', 'melodic'}, Optional*) – The new submode of the relative minor *Key*.

Raises

- *ModeError* – If the *Key* is not 'major'/ 'ionian'.
- SyntaxError – If the *submode* is invalid.

Examples

```
>>> KE = KeyEditor()
>>> key = KE.create_key("D", "major")
>>> KE.relative_minor(key)
D natural minor
>>> key2 = KE.create_key("E", "major")
>>> KE.relative_minor(key, "melodic")
E melodic minor
```

to_roman(*chord*, *scale_key*)
Converts a *Chord* to *Roman*.

Creates the *Roman* based on the *Chord* and a *Scale* or *Key*.

Parameters

- **chord** (Chord) – The *Chord* to be converted.
- **scale_key** (*Scale or Key*) – The *Scale* or *Key* to base the *Roman* on.

Returns The *Roman* of the *Chord*.

Return type *Roman*

Warns **UserWarning** – If the *Chord* is a power or sus chord.

Examples

```
>>> KE = KeyEditor()
>>> SE = ScaleEditor()
>>> CE = ChordEditor()
>>> CRC = ChordRomanConverter()
>>> c_key = KE.create_key("C")
>>> c_scale = SE.create_scale(c_key)
>>> d = CE.create_diatonic(c_scale, 2)
>>> CRC.to_roman(d, c_key)
ii roman chord
>>> f = CE.create_diatonic(c_scale, 4)
>>> CRC.to_roman(f, c_scale)
IV roman chord
```

1.5.2 Musical Classes

- *Note*
- *Key*
- *Scale*
- *Chord*
- *Roman*

Note

class chordparser.**Note**(*letter*, *symbol*)

A class representing a musical note.

The *Note* class consists of notation A-G with optional unicode accidental symbols , , , or . It is created by the *NoteEditor*. When printed, only the *value* of the *Note* is displayed.

Parameters

- **letter** (*str*) – The letter part of the *Note*'s notation. Consists of A-G.
- **symbol** (*str*) – The accidental part of the *Note*'s notation. Consists of the unicode characters , , , or . If there are no accidentals, it is an empty string.

letter

The letter part of the *Note*'s notation.

Type str

symbol

The accidental part of the *Note*'s notation.

Type str

__eq__(*other*)

Compare between other *Notes* and strings.

Checks if the other *Note*'s value or the string is the same as this *Note*.

Parameters **other** – The object to be compared with.

Returns The outcome of the *value* comparison.

Return type boolean

Examples

```
>>> NE = NoteEditor()
>>> d = NE.create_note("D")
>>> d2 = NE.create_note("D")
>>> d_str = "D"
>>> d == d2
True
>>> d == d_str
True
```

Note that symbols are converted to their unicode characters when a *Note* is created.

```
>>> NE = NoteEditor()
>>> ds = NE.create_note("D#")
>>> ds_str = "D#"
>>> ds_str_2 = "D"
>>> ds == ds_str
False
>>> ds == ds_str_2
True
```

accidental(*value*)

Change a *Note*'s accidental by specifying a *value* from -2 to 2.

The range of *values* [-2, 2] correspond to the values a symbol can take, from doubleflat (-2) to doublesharp (2).

Parameters **value** (*int*) – The accidental's integer value.

Raises ValueError – If *value* is not in the range of [-2, 2].

Examples

```
>>> NE = NoteEditor()
>>> d_sharp = NE.create_note("D#")
>>> d_sharp.accidental(-1)
D note
```

letter_value()

Return the *Note*'s letter as an integer value (basis: C = 0).

The value is based on the number of scale degrees above C.

Returns The letter's value.

Return type int

Examples

```
>>> NE = NoteEditor()
>>> d = NE.create_note("D")
>>> d.letter_value()
1
```

num_value()

Return the *Note*'s numerical value (basis: C = 0).

The numerical value is based on the number of semitones above C.

Returns The numerical value.

Return type int

Examples

```
>>> NE = NoteEditor()
>>> d = NE.create_note("D")
>>> d.num_value()
2
```

shift_l(*value*)
Shift a *Note*'s letter.

The *value* corresponds to the change in scale degree of the *Note*.

Parameters **value** (*int*) – The value of the letter shift.

Examples

```
>>> NE = NoteEditor()
>>> d_sharp = NE.create_note("D#")
>>> d_sharp.shift_l(3)
G note
```

shift_s(*value*)
Shift a *Note*'s accidental.

The *Note*'s *symbol_value()* must be in the range of [-2, 2] after the shift, which corresponds to the values a symbol can take from doubleflat (-2) to doublesharp (2).

Parameters **value** (*int*) – The value of the shift in accidentals.

Raises ValueError – If the *Note*'s *symbol_value()* is not in the range of [-2, 2] after the shift.

Examples

```
>>> NE = NoteEditor()
>>> d_sharp = NE.create_note("D#")
>>> d_sharp.shift_s(-1)
D note
```

symbol_value()
Return the *Note*'s symbol as an integer value (basis: natural = 0).

The value is based on the number of semitones away from the natural *Note*.

Returns The symbol's value.

Return type int

Examples

```
>>> NE = NoteEditor()
>>> d_sharp = NE.create_note("D#")
>>> d_sharp.symbol_value()
1
```

transpose(*semitones*, *letters*)
Transpose a *Note* according to semitone and letter intervals.

Parameters

- **semitones** – The difference in semitones to the new transposed *Note*.
- **letters** – The difference in scale degrees to the new transposed *Note*.

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> c.transpose(6, 3)
F note
>>> c.transpose(0, 1)
G note
```

transpose_simple(*semitones*, *use_flats=False*)

Transpose a *Note* according to semitone intervals.

Parameters

- **semitones** (*int*) – The difference in semitones to the new transposed *Note*.
- **use_flats** (*boolean, Optional*) – Selector to use flats or sharps for black keys. Default False when optional.

Examples

```
>>> NE = NoteEditor()
>>> c = NE.create_note("C")
>>> c.transpose_simple(6)
F note
>>> c.transpose(2, use_flats=True)
A note
```

value

The full notation of the *Note*.

Type str

Key

class chordparser.**Key**(*root*, *mode*, *submode*)

A class representing a musical key.

The *Key* class composes of a *Note* class as its *root* as well as the attributes *mode* and *submode*. It is created by the *KeyEditor*. *Keys* can use the same methods as *Notes* to manipulate their *root*.

Parameters

- **root** (Note) – The root note of the *Key*.
- **mode** (*{'major', 'minor', 'ionian', 'dorian', 'phrygian', 'lydian', 'mixolydian', 'aeolian', 'locrian'}*) – The mode of the *Key*.
- **submode** (*{None, 'natural', 'harmonic', 'melodic'}*) – The submode of the *Key*. If the *mode* is not 'minor'/ 'aeolian', *submode* is None. Else, *submode* is one of the strings.

root
The root note of the *Key*.

Type *Note*

mode
The mode of the *Key*.

Type {'major', 'minor', 'ionian', 'dorian', 'phrygian', 'lydian', 'mixolydian', 'aeolian', 'locrian'}

submode
The submode of the *Key*. If the *mode* is not 'minor'/ 'aeolian', *submode* is None. Else, *submode* is one of the strings.

Type {None, 'natural', 'harmonic', 'melodic'}

__eq__(*other*)
Compare between other *Keys*.

Checks if the other *Key* has the same attributes.

Parameters **other** – The object to be compared with.

Returns The outcome of the comparison.

Return type boolean

Examples

```
>>> KE = KeyEditor()
>>> d = KE.create_key("D", "minor")
>>> d2 = KE.create_key("D", "minor", "natural")
>>> d == d2
True
>>> d3 = KE.create_key("D", "minor", "harmonic")
>>> d == d3
False
```

Note that the major mode is the same as ionian, and the minor mode is the same as aeolian.

```
>>> KE = KeyEditor()
>>> e = KE.create_key("E", "major")
>>> e2 = KE.create_key("E", "ionian")
>>> e == e2
True
>>> f = KE.create_key("F", "minor")
>>> f2 = KE.create_key("F", "aeolian")
>>> f == f2
True
```

__getattr__(*attribute*)
Allow *Note* methods to be used on the *Key*'s *root*.

See also:

chordparser.Note() For a list of *Note* methods.

Scale

class chordparser.**Scale**(*key*)
A class representing a musical scale.

The *Scale* composes of a *Key* on which it is based on, and a tuple of *Notes* as part of its *notes* attribute.

Parameters **key** (Key) – The *Key* which the *Scale* is based on.

key
The *Key* which the *Scale* is based on.

Type *Key*

notes
A two-octave tuple of *Notes* of the *Scale*.

Type tuple

scale_intervals
The semitone intervals between *notes*.

Type tuple

__eq__(*other*)
Compare between other *Scales*.

Checks if the other *Scale* has the same *Key* and *notes*.

Parameters **other** – The object to be compared with.

Returns The outcome of the *value* comparison.

Return type boolean

Examples

```
>>> SE = ScaleEditor()
>>> d = SE.create_scale("D", "minor")
>>> d2 = SE.create_scale("D", "minor")
>>> d == d2
True
>>> d3 = SE.create_scale("D", "minor", "harmonic")
>>> d == d3
False
```

build()
Build the *Scale* from its *Key*.

This method does not need to be used if *Scale* adjustments are done through the proper channels (i.e. *ScaleEditor* or using other *Scale* methods), since those would build the *Scale* automatically.

transpose(*semitones*, *letter*)
Transpose a *Scale* according to semitone and letter intervals.

Parameters

- **semitones** – The difference in semitones to the new transposed *root* of the *Scale*'s *Key*.
- **letters** – The difference in scale degrees to the new transposed *root* of the *Scale*'s *Key*.

Examples

```
>>> SE = ScaleEditor()
>>> c = SE.create_scale("C", "major")
>>> c.transpose(6, 3)
F major scale
>>> c.transpose(0, 1)
G major scale
```

transpose_simple(*semitones*, *use_flats=False*)

Transpose a *Scale* according to semitone intervals.

Parameters

- **semitones** (*int*) – The difference in semitones to the new transposed *root* of the *Scale*'s *Key*.
- **use_flats** (*boolean, Optional*) – Selector to use flats or sharps for black keys. Default False when optional.

Examples

```
>>> SE = ScaleEditor()
>>> c = SE.create_scale("C", "minor")
>>> c.transpose_simple(6)
F natural minor scale
>>> c.transpose(2, use_flats=True)
A natural minor scale
```

Chord

class chordparser.**Chord**(*root*, *quality*, *add=None*, *bass=None*, *string=None*)

A musical class representing a chord.

The *Chord* is composed of a *root Note*, *quality*, optional *add Notes* and an optional *bass Note*. It automatically builds its *notes* from these components. When printed, a standardised short notation meant for chord sheets is displayed.

Parameters

- **root** (Note) – The root note.
- **quality** (Quality) – The *Chord* quality.
- **add** (*list of (str, int), Optional*) – List of added notes. The *str* is the accidental and the *int* is the scale degree of each added note.
- **bass** (Note, *Optional*) – Bass note.
- **string** (*str, Optional*) – The *Chord* notation string input.

root

The root note.

Type *Note*

quality

The *Chord* quality.

Type *Quality*

add
List of added notes.

Type list of (str, int), Optional

bass
Bass note.

Type *Note*, Optional

string
The *Chord* notation string input.

Type str, Optional

base_intervals
The intervals of the *Chord* solely based on its *quality*.

Type tuple of int

base_degrees
The scale degrees of the *Chord* solely based on its *quality*.

Type tuple of int

base_symbols
The accidentals of the *Chord* solely based on its *quality*.

Type tuple of str

intervals
The intervals of the *Chord*.

Type tuple of int

degrees
The scale degrees of the *Chord*.

Type tuple of int

symbols
The accidentals of the *Chord*.

Type tuple of str

notes
The tuple of *Notes* in the *Chord*.

Type tuple of Note

__eq__(*other*)
Compare between other *Chords*.

Checks if the other *Chord* has the same attributes. Since the attributes and not the notation is being compared, *Chords* with different notation but same structure are equal (see `Examples`).

Parameters **other** – The object to be compared with.

Returns The outcome of the *value* comparison.

Return type boolean

Examples

```
>>> CE = ChordEditor()
>>> d = CE.create_chord("Dsus")
>>> d2 = CE.create_chord("Dsus4")
>>> d == d2
True
>>> d3 = CE.create_chord("Dsus2")
>>> d == d3
False
```

Another example of the same *Chord* with different notation:

```
>>> CE = ChordEditor()
>>> e = CE.create_chord("Eaug7")
>>> e2 = CE.create_chord("E7#5")
>>> e == e2
True
```

build()
Build the *Chord* from its attributes.

This method does not need to be used if *Chord* adjustments are done through the proper channels (i.e. *ChordEditor* or using other *Chord* methods), since those would build the *Chord* automatically.

transpose(*semitones*, *letter*)
Transpose a *Chord* according to semitone and letter intervals.

Parameters

- **semitones** – The difference in semitones to the new transposed *root* of the *Chord*.
- **letters** – The difference in scale degrees to the new transposed *root* of the *Chord*.

Examples

```
>>> CE = ChordEditor()
>>> c = CE.create_chord("Csus")
>>> c.transpose(6, 3)
Fsus chord
>>> c.transpose(0, 1)
Gsus chord
```

transpose_simple(*semitones*, *use_flats=False*)
Transpose a *Chord* according to semitone intervals.

Parameters

- **semitones** (*int*) – The difference in semitones to the new transposed *root* of the *Chord*.
- **use_flats** (*boolean, Optional*) – Selector to use flats or sharps for black keys. Default False when optional.

Examples

```
>>> CE = ChordEditor()
>>> c = CE.create_chord("Cm")
>>> c.transpose_simple(6)
Fm chord
>>> c.transpose(2, use_flats=True)
Am chord
```

Roman

class chordparser.**Roman**(*root*, *quality*, *inversion*)

A class representing Roman numeral notation.

The *Roman* is composed of its *root*, *quality* and *inversion*. When printed, the standard Roman numeral notation is displayed.

Parameters

- **root** (*str*) – The scale degree of the *Roman*. Uppercase if major/augmented and lowercase if minor/diminished.
- **quality** (*str*) – The quality of the *Roman*.
- **inversion** (*tuple of int*) – The inversion of the *Roman* in figured bass notation (e.g. (6, 4) for second inversion).

root

The scale degree of the *Roman*. Uppercase if major/augmented and lowercase if minor/diminished.

Type str

quality

The quality of the *Roman*.

Type str

inversion

The inversion of the *Roman* in figured bass notation (e.g. (6, 4) for second inversion).

Type tuple of int

__eq__(*other*)

Compare between other *Romans*.

Checks if the other *Roman* has the same *root*, *quality* and *inversion*.

Parameters **other** – The object to be compared with.

Returns The outcome of the *value* comparison.

Return type boolean

Examples

```
>>> KE = KeyEditor()
>>> SE = ScaleEditor()
>>> CE = ChordEditor()
>>> CRC = ChordRomanConverter()
>>> c_key = KE.create_key("C")
>>> c_scale = SE.create_scale(c_key)
```

(continues on next page)

(continued from previous page)

```
>>> d = CE.create_diatonic(c_scale, 2)
>>> r = CRC.to_roman(d, c_key)
>>> r2 = CRC.to_roman(d, c_key)
>>> r == r2
True
>>> e = CE.create_diatonic(c_scale, 3)
>>> r3 = CRC.to_roman(e, c_key)
>>> r == r3
False
```

1.5.3 Helper Classes

- *Quality*

Quality

class chordparser.**Quality**(*quality_str*, *ext_str=None*, *flat_ext=False*)

A class representing the quality of a *Chord*.

The *Quality* class composes of its base *Chord* quality, extensions to the *Chord*, and optional flats on the extended *Note*.

Parameters

- **quality_str** (*str*) – The *Quality*'s notation.
- **ext_str** (*str, Optional*) – The extended *Chord*'s notation.
- **flat_ext** (*boolean, Optional*) – Selector for the flat of the extended *Note*. Default False when optional.

value

The *Quality*'s notation.

Type str

ext

The extended *Chord*'s notation.

Type str, Optional

flat_ext

Selector for the flat of the extended *Note*. Default False when optional.

Type boolean, Optional

base_intervals

The intervals of the triad of a *Chord* with this *Quality*.

Type tuple of int

base_degrees

The scale degrees of the triad of a *Chord* with this *Quality*.

Type tuple of int

base_symbols
The accidentals of the triad of a *Chord* with this *Quality*.

Type tuple of str

intervals
The intervals of a *Chord* with this *Quality*.

Type tuple of int

degrees
The scale degrees of a *Chord* with this *Quality*.

Type tuple of int

symbols
The accidentals of a *Chord* with this *Quality*.

Type tuple of str

Raises

- `ValueError` – If a dominant chord has a *value* of 'major' (*value* should be 'dominant')
- `ValueError` – If a diminished extended chord does not have a *ext* of 'diminished seventh'
- `ValueError` – If a 'seventh' extended chord has a flat extension (should be reflected in *ext* and not in *flat_ext*)

__eq__(*other*)
Compare between other *Qualitys*.

Checks if the other *Quality* has the same *value*, *ext* and *flat_ext*.

Parameters **other** – The object to be compared with.

Returns The outcome of the *value* comparison.

Return type boolean

Examples

```
>>> QE = QualityEditor()
>>> q = QE.create_quality("maj9")
>>> q2 = QE.create_quality("maj9")
>>> q == q2
True
>>> q3 = QE.create_quality("majb9")
>>> q == q3
False
```

1.5.4 Errors

- *ModeError*

ModeError

class chordparser.**ModeError**
Exception where a *Key*'s *mode* is invalid for an operation.

1.6 Background

Work in progress.

1.7 Credits

1.7.1 Development Lead

- Titus Ong <titusongyl@gmail.com>

1.7.2 Contributors

- amelie106

1.8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.8.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/titus-ong/chordparser/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

chordparser could always use more documentation, whether as part of the official chordparser docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/titus-ong/chordparser/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.8.2 Get Started!

Ready to contribute? Here's how to set up chordparser for local development.

1. Fork the chordparser repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:titus-ong/chordparser.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv chordparser
$ cd chordparser/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 chordparser tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

1.8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, 3.7 and 3.8, and for PyPI. Check https://travis-ci.com/titus-ong/chordparser/pull_requests and make sure that the tests pass for all supported Python versions.

1.9 Licence

1.10 History

1.10.1 0.4.2 (2023-03-26)

- Fix incorrect roman scale conversion

1.10.2 0.4.1 (2020-07-28)

- Fix imports for autodoc to show documentation properly

1.10.3 0.4.0 (2020-07-28)

- Reorganised documentation to follow Divio documentation
- Updated README with quick demo
- Completed Colab notebook as a tutorial and showcase
- Included first draft of reference documentation in NumPy documentation style
- Catch errors in Parser if `Parser.sample` fails to be read

- Changed some keyword argument names to better fit with documentation

1.10.4 0.3.11 (2020-07-23)

- Updated `analyse_secondary` method to be limited to secondary dominants and secondary leading tone chords by default
- Fixed `analyse_secondary` method to return an empty string if the next chord is the tonic

1.10.5 0.3.10 (2020-07-23)

- Include `analyse_secondary` method

1.10.6 0.3.9 (2020-07-23)

- Fix accidental print in NoteEditor
- Include `__str__` representation for printing musical classes
- Fix file reading for Colab notebook

1.10.7 0.3.8 (2020-07-23)

- Reorganised code folder structure
- Changed note structure to be split into letter and symbol
- Added `get_letter_intervals` method to NoteEditor
- Refactored tests to have more of a single responsibility

1.10.8 0.3.7 (2020-07-20)

- Fixed empty README author (fixes failed build - PyPI could not read the README)

1.10.9 0.3.6 (2020-07-20)

- **FAILED BUILD**
- Fixed empty README description
- Fixed Sphinx documentation for readthedocs

1.10.10 0.3.5 (2020-07-20)

- **FAILED BUILD**
- Included `transpose_simple` method to only use semitones for transposing
- Included initial Sphinx documentation

1.10.11 0.3.4 (2020-07-19)

- Removed code with static typing
- Removed unused instance variables in classes
- Changed `Note.symbolvalue()` to `Note.symbol_value()` for namespace consistency
- Include `Note.letter_value()` method
- Fixed wrong errors being raised
- Refactored scale building code
- Abstracted chord quality to a separate Quality class
- Quality now includes sus chords, Chords no longer have a `Chord.sus` attribute
- Refactored chord notation regex parsing for methods to be reusable
- Abstracted roman numeral notation to a separate Roman class, and roman conversion to a separate ChordRomanConverter class

1.10.12 0.2.2 (2020-06-25)

- Changed song in sample_sheet.cho
- Added ability to remove all added notes
- `incl_submodes` defaulted to False in analysing chords

1.10.13 0.2.1 (2020-06-24)

- Bug fixes for sample_sheet.cho

1.10.14 0.2.0 (2020-06-24)

- Separated parsing function into Editor classes - NoteEditor, KeyEditor, ScaleEditor, ChordEditor
- Refactored regex parsing for code to be more readable and for regex groups to be more distinct
- Added ChordAnalyser for analysing chords
- Added Parser class to collate all the Editors

1.10.15 0.1.1 (2019-10-21)

- Added Chord class with full parsing function

1.10.16 0.1.0 (2019-10-11)

- First release on PyPI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__eq__() (*chordparser.Chord method*), 23
__eq__() (*chordparser.Key method*), 20
__eq__() (*chordparser.Note method*), 16
__eq__() (*chordparser.Quality method*), 27
__eq__() (*chordparser.Roman method*), 25
__eq__() (*chordparser.Scale method*), 21
__getattr__() (*chordparser.Key method*), 20

A

accidental() (*chordparser.Note method*), 17
add (*chordparser.Chord attribute*), 23
analyse_all() (*chordparser.Parser method*), 5
analyse_diatonic() (*chordparser.Parser method*), 6
analyse_secondary() (*chordparser.Parser method*), 7

B

base_degrees (*chordparser.Chord attribute*), 23
base_degrees (*chordparser.Quality attribute*), 26
base_intervals (*chordparser.Chord attribute*), 23
base_intervals (*chordparser.Quality attribute*), 26
base_symbols (*chordparser.Chord attribute*), 23
base_symbols (*chordparser.Quality attribute*), 26
bass (*chordparser.Chord attribute*), 23
build() (*chordparser.Chord method*), 24
build() (*chordparser.Scale method*), 21

C

change_chord() (*chordparser.Parser method*), 8
change_key() (*chordparser.Parser method*), 9
change_note() (*chordparser.Parser method*), 9
change_scale() (*chordparser.Parser method*), 10
Chord (*class in chordparser*), 22
create_chord() (*chordparser.Parser method*), 10
create_diatonic() (*chordparser.Parser method*), 11
create_key() (*chordparser.Parser method*), 11
create_note() (*chordparser.Parser method*), 12
create_scale() (*chordparser.Parser method*), 12

D

degrees (*chordparser.Chord attribute*), 23
degrees (*chordparser.Quality attribute*), 27

E

ext (*chordparser.Quality attribute*), 26

F

flat_ext (*chordparser.Quality attribute*), 26

G

get_intervals() (*chordparser.Parser method*), 13
get_letter_intervals() (*chordparser.Parser method*), 13
get_min_intervals() (*chordparser.Parser method*), 13
get_tone_letter() (*chordparser.Parser method*), 14

I

intervals (*chordparser.Chord attribute*), 23
intervals (*chordparser.Quality attribute*), 27
inversion (*chordparser.Roman attribute*), 25

K

key (*chordparser.Scale attribute*), 21
Key (*class in chordparser*), 19

L

letter (*chordparser.Note attribute*), 16
letter_value() (*chordparser.Note method*), 17

M

mode (*chordparser.Key attribute*), 20
ModeError (*class in chordparser*), 28

N

Note (*class in chordparser*), 16
notes (*chordparser.Chord attribute*), 23
notes (*chordparser.Scale attribute*), 21
num_value() (*chordparser.Note method*), 17

P

Parser (*class in chordparser*), 4

Q

quality (*chordparser.Chord attribute*), 22
quality (*chordparser.Roman attribute*), 25
Quality (*class in chordparser*), 26

R

relative_major() (*chordparser.Parser method*), 14
relative_minor() (*chordparser.Parser method*), 14
Roman (*class in chordparser*), 25
root (*chordparser.Chord attribute*), 22
root (*chordparser.Key attribute*), 19
root (*chordparser.Roman attribute*), 25

S

Scale (*class in chordparser*), 21
scale_intervals (*chordparser.Scale attribute*), 21
shift_l() (*chordparser.Note method*), 18
shift_s() (*chordparser.Note method*), 18
string (*chordparser.Chord attribute*), 23
submode (*chordparser.Key attribute*), 20
symbol (*chordparser.Note attribute*), 16
symbol_value() (*chordparser.Note method*), 18
symbols (*chordparser.Chord attribute*), 23
symbols (*chordparser.Quality attribute*), 27

T

to_roman() (*chordparser.Parser method*), 15
transpose() (*chordparser.Chord method*), 24
transpose() (*chordparser.Note method*), 18
transpose() (*chordparser.Scale method*), 21
transpose_simple() (*chordparser.Chord method*), 24
transpose_simple() (*chordparser.Note method*), 19
transpose_simple() (*chordparser.Scale method*), 22

V

value (*chordparser.Note attribute*), 19
value (*chordparser.Quality attribute*), 26