
babs Documentation

iskyd

Mar 28, 2019

Contents

1	Installation	3
1.1	Test	3
1.2	Coverage	3
2	Note	5
2.1	Create your first note	5
2.2	Octave	6
2.3	Alteration	6
2.4	Duration	6
2.5	Change attribute	7
2.6	Pitch shift	7
2.7	__str__ and __repr__	8
2.8	Comparison	9
3	Rest	11
3.1	__repr__	11
3.2	Comparison	11
4	NoteList	13
4.1	Create your first NoteList	13
4.2	strict	13
4.3	notes attribute	14
4.4	add note	14
4.5	remove note	15
4.6	is valid	16
4.7	Comparison	16
4.8	get_notes_from_root	16
5	Chord	19
5.1	Create your first chord	19
5.2	strict	20
5.3	notes attribute	20
5.4	add note	20
5.5	remove note	22
5.6	is valid	23
5.7	Comparison	23
5.8	Create from root Note	23

5.9	List of pre-defined chord type	24
6	Scale	25
6.1	Create your first scale	25
6.2	strict	26
6.3	notes attribute	26
6.4	add note	27
6.5	remove note	27
6.6	is valid	29
6.7	Comparison	29
6.8	Create Scale from root Note	29
6.9	List of pre-defined scale type	30
7	Authors	31
8	Indices and tables	33

Contents:

CHAPTER 1

Installation

Download babs

```
git clone https://github.com/iskyd/babs
cd babs
```

Create virtual environment (OPTIONALS)

```
python -m venv venv
```

If you are using python you can use virtualenv

Install babs

```
pip install -e .
```

1.1 Test

```
pip install pytest
pytest
```

Run single test

```
pytest tests/[test_file.py]
```

1.2 Coverage

```
pip install pytest-cov
pytest --cov=babs --cov-report html:htmlcov
```

It will create htmlcov directory with html report.

If everything goes well... You are ready to get started!


```
class Note (freq, name, octave=4, alt='sharp', duration=4/4)
    Return a musical Note representation from frequency or name

    pitch_shift (val[, half_step=False, octave=False, alt=None ])
        change the frequency of the current note

    get_note_index ()
        return the note index in Note.NOTES or False if not found

    classmethod get_note_name_by_index (idx[, alt=None ])
        return the name of the note at specified idx in Note.NOTES. idx could be any number so that you don't
        have to worry about idx > len(Note.NOTES)
```

2.1 Create your first note

```
from babs import Note

n = Note(freq=440)
```

Now you have create a Note with 440Hz of frequency (the A at 4th octave). So you now have access to attribute's note.

```
print (n.freq)    # 440
print (n.name)    # 'A'
print (n.octave)  # 4
print (n.duration) # 1.0 (4/4)
```

You can also create a note starting from it name.

```
n = Note(name='A')

print (n.freq)    # 440
```

(continues on next page)

(continued from previous page)

```
print(n.name) # 'A'
print(n.octave) # 4
print(n.duration) # 1.0 (4/4)
```

The freq params have the priority so if you specify the frequency, name and octave will be ignored:

```
n = Note(freq=440, name='B', octave=3)

print(n.freq) # 440
print(n.name) # 'A'
print(n.octave) # 4
```

2.2 Octave

The octave is note's position on a standard 88-key piano keyboard and by default (as you can see in the previous example) is 4.

```
n = Note(name='A', octave=5)

print(n.freq) # 880.0
print(n.name) # 'A'
print(n.octave) # 5
print(n.duration) # 1.0 (4/4)
```

2.3 Alteration

Now let's create a Bb note.

```
n = Note(freq=466.16)
print(n.name) # 'A#'
```

So we got A# as name because the A# and the Bb note has the same frequency. But what if you need to get a Bb? You can use alteration attribute.

```
n = Note(freq=466.16, alt='flat')

print(n.name) # 'Bb'
```

2.4 Duration

The duration represent the relative duration of the note.

```
n = Note(name='A', duration=3/4)

print(n.duration) # 0.75 -> 3/4
```

You can change the duration of the note simply using the setter.

```
print(n.duration) # 0.75 -> 3/4
n.duration = 2/4
print(n.duration) # 0.5 -> 3/4
```

2.5 Change attribute

You can easily change the note frequency and the note alteration. Babs will calculate again the name and the octave of the note.

```
n = Note(freq=440)

print(n.name) # 'A'

n.freq = 466.16
print(n.name) # 'A#'

n.alt = 'flat'
print(n.name) # 'Bb'

n.freq = 880
print(n.name) # 'A'
print(n.octave) # 5
```

2.6 Pitch shift

If you need more control to alter a note then just using the freq setter you can use a more powerful function, the pitch shift. The pitch shift can be used in three different way.

- Add or sub a frequency value

```
n = Note(freq=440)

n.pitch_shift(value=26.16) # Increase the freq by 26.16hz
print(n.freq) # 466.16
print(n.name) # 'A#'

n.pitch_shift(value=-26.16) # Decrease the freq by 26.16hz
print(n.freq) # 440.0
print(n.name) # 'A'
```

- Add or sub an octave value

```
n = Note(freq=440)

n.pitch_shift(value=2, octave=True) # Add 2 octaves
print(n.freq) # 1760.0
print(n.name) # 'A'
print(n.octave) # 6

n.pitch_shift(value=-3, octave=True) # Sub 3 octaves
print(n.freq) # 220.0
print(n.name) # 'A'
print(n.octave) # 3
```

- Add or sub an half tone value

```
n = Note(freq=440)

n.pitch_shift(value=2, half_step=True) # Add 1 tone (2 half tones)
print(n.freq) # 493.88
print(n.name) # 'B'
print(n.octave) # 4

n.pitch_shift(value=-12, half_step=True) # Sub 6 tones (6 tones = 1 octave)
print(n.freq) # 246.94
print(n.name) # 'B'
print(n.octave) # 3
```

With `half_step` and `octave` you can specify the alteration you need as before

```
n = Note(freq=440)

n.pitch_shift(value=1, half_step=True, alt='flat') # Add half tone
print(n.freq) # 466.16
print(n.name) # 'Bb'
print(n.octave) # 4
```

Remember that 0 is a valid value so the following will work:

```
n = Note(freq=466.16)

n.pitch_shift(value=0, alt='flat') # Add half tone
print(n.freq) # 466.16
print(n.name) # 'Bb'
print(n.octave) # 4
```

Consider that we can obtain the same result in this **recommended** way without using the pitch shift function:

```
n = Note(freq=466.16)

n.alt = 'flat' # Add half tone
print(n.freq) # 466.16
print(n.name) # 'Bb'
print(n.octave) # 4
```

2.7 `__str__` and `__repr__`

`__str__` will return the current name and octave of the note.

```
str(Note(freq=440)) # 'A4'
```

`__repr__` will return the current representation of the Note so that you can call `eval()` on it.

```
repr(Note(freq=440)) # 'Note(freq=440, alt='None', duration=1.0)'

n = Note(freq=440, duration=1/8) # repr(n) -> Note(freq=440, alt='None', duration=0.
↪ 125)
x = eval(repr(x)) # x will be the same as n
```

2.8 Comparison

Note support equal and not equal comparison operator. Two notes are the same if they have the same frequency and the same duration.

```
Note(name='A') == Note(name='A')      # True
Note(name='A') == Note(name='A', duration=1/8)  # False
Note(name='A') != Note(name='C')      # True
Note(name='A') != Note(name='A', duration=1/8)  # True
```



```
class Rest (duration=4/4)
```

Return a rest, an interval of silence. duration represent the relative duration of the rest.

3.1 `__repr__`

`__repr__` will return the current representation of the Rest so that you can call `eval()` on it.

```
repr(Rest(duration=1/4))  # 'Rest(duration=0.25) '  
  
r = Rest(duration=1/4)  
x = eval(repr(r))  # x will be the same as r
```

3.2 Comparison

Rest support all comparison operator. This is just a shortcut of using `.duration` attribute.

```
Rest(duration=1/8) < Rest(duration=1/4) == Rest(duration=1/8).duration < Rest(duration=1/4).duration  # True
```


4.1 Create your first NoteList

NoteList is an abstract class, so you can't instantiate it. We create a Mock class that extends NoteList in order to instantiate it.

```
from babs import Note, NoteList

class Mock(NoteList):
    pass

m = Mock(Note(name='C'), Note(name='D'), Note(name='E'))
```

Now you have create the C chord and you can access to the notes attribute.

```
print(len(m.notes))
for n in m.notes:
    print(n)

# 3
# C4
# D4
# E4
```

4.2 strict

As we said a NoteList consists of one or more note. So if we create an empty NoteList we'll get a NoteListException.

```
from babs import Note, NoteList

class Mock(NoteList):
```

(continues on next page)

(continued from previous page)

```
pass

m = Mock()

# NoteListException "Invalid notes given."
```

If you need to create an “invalid” NoteList you can use strict parameter.

```
m = Mock(strict=False)
print(len(m.notes)) # 0
```

If you set strict to False you can also pass an invalid Note to NoteList.

```
m = Mock('invalid', 'notes', strict=False)
print(len(m.notes)) # 3
```

4.3 notes attribute

You can only get the notes attribute but you can’t set it!

```
m = Mock(strict=False)

m.notes = [Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=5)]

# AttributeError: can't set attribute
```

4.4 add note

You can use this method if you need to add a Note to the current list.

```
m = Mock(Note(name='C'), Note(name='D'), Note(name='C'))

c.add_note(note=Note(name='E'))

print(len(c.notes)) # 4

for n in c.notes:
    print(n)

# C4
# D4
# C4
# E4
```

By default strict is set to True, so if you try to add an invalid Note you will get a NoteListException

```
m.add_note('invalid') # Add a string instead of a Note

# NoteListException: Instance of Note expected, str given.

m = Mock(Note(name='C'), Note(name='D'))
```

(continues on next page)

(continued from previous page)

```
m.add_note('invalid', strict=False)

print(len(m.notes))  # 3
```

4.5 remove note

If you need to remove a Note you can use the `remove_note()` method. You can remove a note by `Note()`, name, frequency or octave.

```
m = Mock(Note(name='C'), Note(name='E'), Note(name='G'))
c.remove_note(note=Note(name='G'))

print(len(m.notes))  # 2

for n in m.notes:
    print(n)

# C4
# E4
```

By default, as before, `strict` is set to `True`, so if the list will be invalid after remove you will have a `NoteListException`. If a `NoteListException` is raised the notes in the list will be restored as they were before the remove.

```
m = Mock(Note(name='C'))

m.remove_note(note=Note(name='C'))
# Invalid Mock.

print(len(c.notes))  # 1
```

Removing a Note by octave or name can remove multiple notes.

```
m = Mock(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=5))

print(len(m.notes))  # 4

m.remove_note(name='C')
print(len(m.notes))  # 2

for n in m.notes:
    print(n)

# E4
# G4

m = Mock(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=5))
m.remove_note(octave=4)

print(len(m.notes))  # 1

for n in m.notes:
    print(n)

# C5
```

4.6 is valid

If you need to know if the actual list is valid you can use `is_valid` method. A `NoteList` is valid if has one or more notes and if all notes are valid `Note` object (instance of `Note`)

4.7 Comparison

`NoteList` support equal and not equal comparison operator. Two `NoteList` are the same if they have the same notes (check note comparison for more details). The `strict` attribute in `create` doesn't affect the list comparison.

```
Mock(Note(name='A'), Note(name='C')) == Mock(Note(name='A'), Note(name='C')) # True
Mock(Note(name='A'), Note(name='C'), strict=True) == Mock(Note(name='A'), Note(name='C',
↳), strict=False) # True
Mock(Note(name='A'), Note(name='C')) == Mock(Note(name='A'), Note(name='C'),
↳Note(name='E')) # False
Mock(Note(name='A'), Note(name='C')) != Mock(Note(name='A'), Note(name='C')) # False
```

4.8 get_notes_from_root

You can easily get a list of notes from a root chord using the `get_notes_from_root` classmethod. Suppose you want to get a basic list of note C - E - G.

```
notes = Mock.get_notes_from_root(root=Note(name='C'), note_list_type=[4, 7])
for n in notes:
    print(n)

# C4
# E4
# G4
```

That's it, you've got a C - E - G list. So how it works? `get_notes_from_root` use a `note_list_type`, which is a list of notes distance from root note. So the [4, 7]. 4 is the distance between root from 3d and between the root and the 5th. The distance is based on the `Note.NOTES` list: `NOTES = ['C', 'C#/Db', 'D', 'D#/Eb', 'E', 'F', 'F#/Gb', 'G', 'G#/Ab', 'A', 'A#/Bb', 'B']` So let's say we want a major seven chord.

```
notes = Mock.get_notes_from_root(root=Note(name='C'), note_list_type=[4, 7, 11])
for n in notes:
    print(n)

# C4
# E4
# G4
# B4
```

This works because the index the E is distant 4 elements from the root (C) in the `Note.NOTES` list, the G is distant 7 elements and the B is distant 11 elements. By default the octave of other notes will be the same as the root note. To change that behaviour you can use the `octave` param. **The root note will not be affected by the octave param.** octave could be :

- an integer, so that every note will have that specific octave.

```

notes = Mock.get_notes_from_root(root=Note(name='C'), note_list_type=[2, 4], octave=3)

for n in notes:
    print(n)

# C4
# D3
# E3

```

- a string ‘root’ `NoteList.OCTAVE_TYPE_ROOT` or ‘from_root’ `NoteList.OCTAVE_TYPE_FROM_ROOT`:

- ‘root’, the default behaviour, use the root.octave.
- ‘from_root’ use the root.octave as the starting octave. So if you have a G-B-D, the starting octave is 4 so we have a G4, then we have a B4 and at least the D Note goes to the next octave so it is a D5.

```

notes = NoteList.get_notes_from_root(root=Note(name='G'), note_list_type=[4, 7],
↳octave=NoteList.OCTAVE_TYPE_FROM_ROOT)

for n in notes:
    print(n)

# G4
# B4
# D5

```

- a callable, that must return an integer. In the callable you have access to root_octave, i (the idx of list distance iteration, starting from 0) and the distance between the current note and the root note. So suppose you want to have a list that looks like G4, B5, D6.

```

notes = NoteList.get_notes_from_root(
    root=Note(name='G'),
    note_list_type=[4, 7], octave=lambda root_octave, i, distance: root_octave +
↳i + 1
)

for n in notes:
    print(n)

# G4
# B5
# D6

```

If the octave is invalid, for example a string different from ‘root’ and ‘from_root’ or a float number, it will be set to 4.

You can also specify the alteration of the notes using the alt param (by default is ‘sharp’ `Note.SHARP`) that works in the same way as for single note.

```

notes = NoteList.get_notes_from_root(root=Note(name='G'), note_list_type=[4, 7, 11],
↳alt=Note.SHARP)

for n in notes:
    print(n)

# G4
# B4
# D4

```

(continues on next page)

(continued from previous page)

```
# F#4

notes = NoteList.get_notes_from_root(root=Note(name='G'), note_list_type=[4, 7, 11],
↪alt=Note.FLAT)

for n in notes:
    print(n)

# G4
# D4
# B4
# Gb4
```

```
class Chord(*notes, **kwargs)
```

Return a Chord, an harmonic set of pitches consisting of two or more notes. Chord is an ordered list of notes from lower to higher. If strict is True (default) raise ChordException if chord has less then two notes or if Notes are not valid Note object.

Chord extends NoteList so it inherits all methods from NoteList: `is_valid()`, `add_note(note[, strict=True])`, `remove_note(note=None, freq=None, name=None, octave=None[, strict=True])`.

```
classmethod create_from_root(root[, chord_type=None, octave='root', alt='sharp'])
Create and return a Chord from the root note
```

5.1 Create your first chord

```
from babs import Note, Chord

c = Chord(Note(name='C'), Note(name='E'), Note(name='G'))
```

Now you have create the C chord and you can access to the notes attribute.

```
for n in c.notes:
    print(n)

# C4
# E4
# G4
```

Chord is an ordered list of notes from lower to higher. So the order in which you pass the notes in the constructor is not relevant, the list will always be from lower note (lower frequency) to the higher note (higher frequency).

```
c = Chord(Note(name='E'), Note(name='G'), Note(name='C'))
```

(continues on next page)

(continued from previous page)

```
for n in c.notes:
    print(n)

# C4
# E4
# G4

c = Chord(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=3))

for n in c.notes:
    print(n)

# C3
# C4
# E4
# G4
```

5.2 strict

As we said a Chord consists of two or more note. So what happen if we create a one Note or an empty Chord?

```
c = Chord(Note(name='C'))
```

ChordException (Invalid Chord.) You can disable is_valid() control passing strict=False. In this way you will also disable the ordering, because order() works with Note object (using the attribute freq of the note) and will raise AttributeError if the object is not a valid Note.

```
c = Chord(Note(name='C'), strict=False)
print(len(c.notes)) # 1
```

If you set strict to False you also disable Note check so this will not raise an exception.

```
c = Chord('a', 'b', 'c', strict=False)
print(len(c.notes)) # 3
```

5.3 notes attribute

You can only get the notes attribute but not set it!

```
c = Chord(Note(name='C'), Note(name='E'))

c.notes = [Note(name='C'), Note(name='E'), Note(name='G'), Note(name='Bb')]

# AttributeError: can't set attribute
```

5.4 add note

This method use the add_note() method of NoteList abstract class but re-order the notes (if Chord is valid) after the note is added.


```

c = Chord(Note(name='C'), Note(name='E'), Note(name='G'))

c.add_note(note=Note(name='Bb'))

print(len(c.notes))    # 4

for n in c.notes:
    print(n)

# C4
# E4
# G4
# Bb4

c.add_note(note=Note(name='C', octave=3))

print(len(c.notes))    # 5

for n in c.notes:
    print(n)

# C3
# C4
# E4
# G4
# Bb4

```

You can add the same note multiple times:

```

c = Chord(Note(name='A', octave=3), Note(name='C'), Note(name='E'))

c.add_note(note=Note(name='E'))

print(len(c.notes))

for n in c.notes:
    print(n)

# A3
# C4
# E4
# E4

```

By default strict is set to True, so if you add an invalid Note you will get a ChordException

```

c = Chord(Note(name='C'), Note(name='E'), Note(name='G')).add_note(note='c')    # Add a_
↪ string instead of a Note

# ChordException: Instance of Note expected, str given.

c = Chord(strict=False)
c.add_note(note='c', strict=False)

print(len(c.notes))    # 1

```

5.5 remove note

This method use the `remove_note()` method of `NoteList` abstract class but re-order the notes (if Chord is valid) after the note is added.

```
c = Chord(Note(name='C'), Note(name='E'), Note(name='G'))
c.remove_note(note=Note(name='G'))

print(len(c.notes))  # 2

for n in c.notes:
    print(n)

# C4
# E4
```

By default, as before, `strict` is set to `True`, so if the Chord will be invalid after remove you will have a `ChordException`. If `ChordException` is raised the notes in the chord will be restored as they were before the remove.

```
c = Chord(Note(name='C'), Note(name='E'))

c.remove_note(note=Note(name='E'))
# Invalid Chord.

print(len(c.notes))  # 2

for n in c.notes:
    print(n)

# C4
# E4
```

Removing a Note by octave or name can remove multiple notes.

```
c = Chord(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=5))

print(len(c.notes))  # 4

c.remove_note(name='C')
print(len(c.notes))  # 2

for n in c.notes:
    print(n)

# E4
# G4

c = Chord(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='C', octave=5))
c.remove_note(octave=4, strict=False)

print(len(c.notes))  # 1

for n in c.notes:
    print(n)

# C5
```

5.6 is valid

If you need to know if the actual Chord is valid you can use `is_valid` method. A chord is valid if has two or more Note and if all notes are instance of Note()

5.7 Comparison

Chord support equal and not equal comparison operator. Check NoteList documentation for more information

5.8 Create from root Note

You can easily create a Chord from root note using the `create_from_root` classmethod. Suppose you want create a C major chord.

```
c = Chord.create_from_root (root=Note (name='C'))
for n in c.notes:
    print (n)

# C4
# E4
# G4
```

That's it, you've got a C major chord. `create_from_root` use the classmethod `get_notes_from_root` of NoteList. Check NoteList documentation for more information.

babs come with some of pre-defined chord_type so that the previous example could be the same as

```
c = Chord.create_from_root (root=Note (name='C'), chord_type=Chord.MAJOR_SEVEN_TYPE)
```

You can use a custom list or use some of the pre-defined chord type.

5.9 List of pre-defined chord type

Chord type
MAJOR_TYPE
MAJOR_SEVEN_TYPE
MINOR_TYPE
MINOR_SEVEN_TYPE
DOMINANT_TYPE
MINOR_MAJOR_SEVEN_TYPE
HALF_DIMINISHED_SEVEN_TYPE
DIMINISHED_TYPE
DIMINISHED_SEVEN_TYPE
AUGMENTED_TYPE
AUGMENTED_SEVEN_TYPE
AUGMENTED_MAJOR_SEVEN_TYPE
MAJOR_SIXTH_TYPE
MINOR_SIXTH_TYPE
SUS4_TYPE
SUS4_SEVEN_TYPE
SUS4_MAJOR_SEVEN_TYPE
SUS2_TYPE
SUS2_SEVEN_TYPE
SUS2_MAJOR_SEVEN_TYPE

CHAPTER 6

Scale

```
class Scale (*notes, **kwargs)
```

Return a Scale, a set of musical notes ordered by fundamental frequency or pitch. Scale is an ordered list of unique notes. If strict is True (default) raise ScaleException if Notes are not valid Note object.

Scale extends NoteList so it inherits all methods from NoteList: `is_valid()`, `add_note(note[, strict=True])`, `remove_note(note=None, freq=None, name=None, octave=None[, strict=True])`.

```
classmethod create_from_root (root[, scale_type=None, octave='root', alt='sharp'])  
Create and return a Scale from the root note
```

6.1 Create your first scale

```
from babs import Note, Scale  
  
s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G'  
→'), Note(name='A'), Note(name='B'))
```

Now you have create the C major scale and you can access to the notes attribute.

```
for n in s.notes:  
    print (n)  
  
# C4  
# D4  
# E4  
# F4  
# G4  
# A4  
# B4
```

Scale is an ordered list of unique notes. Scale could be ascending (from lower to higher) or descending (from higher to lower), by default is ascending. The order in which you pass the notes in the constructor is not relevant, the list will

always be from lower to higher (frequency) or from higher to lower (frequency) note.

```
s = Scale(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='A'), Note(name='F')
↪), Note(name='D'), Note(name='B'))

for n in s.notes:
    print(n)

# C4
# D4
# E4
# F4
# G4
# A4
# B4

s = Scale(Note(name='C'), Note(name='E'), Note(name='G'), Note(name='A'), Note(name='F')
↪), Note(name='D'), Note(name='B'), order=Scale.DESENDING_SCALE_TYPE)

for n in s.notes:
    print(n)

# B4
# A4
# G4
# F4
# E4
# D4
# C4
```

6.2 strict

As we said a Scale is a set of unique notes. So what happen if we create an empty Scale?

```
s = Scale()
```

ScaleException (Invalid Scale.) You can disable `is_valid()` control passing `strict=False`. In this way you will also disable the ordering, because `order()` works with Note object (using the attribute `freq` of the note) and will raise `AttributeError` if the object is not a valid Note.

```
s = Scale(strict=False)
print(len(s.notes)) # 0
```

If you set `strict` to `False` you also disable Note check so this will not raise an exception.

```
s = Scale('a', 'b', 'c', strict=False)
print(len(c.notes)) # 3
```

6.3 notes attribute

You can only get the `notes` attribute but not set it!

```
s = Scale(Note(name='C'))

s.notes = [Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G'), Note(name='A'), Note(name='B')]

# AttributeError: can't set attribute
```

6.4 add note

This method use the `add_note()` method of `NoteList` abstract class but re-order the notes (if `Scale` is valid) after the note is added. It also checks if the note already exists in the scale and raise an exception if it already exists.

```
s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G'), Note(name='A'))

s.add_note(note=Note(name='B'))

print(len(s.notes)) # 7

for n in s.notes:
    print(n)

# C4
# D4
# E4
# F4
# G4
# A4
# B4

s.add_note(note=Note(name='B'))

# babs.exceptions.scale_exception.ScaleException: Note B4 is already in Scale.
```

By default `strict` is set to `True`, so if you add an invalid `Note` or a note that already exists in the scale you will get a `ScaleException`

```
s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G'), Note(name='A'))

s.add_note(note='c') # Add a string instead of a Note

# ScaleException: AttributeError: 'str' object has no attribute 'freq'
```

6.5 remove note

This method use the `remove_note()` method of `NoteList` abstract class but re-order the notes (if `Scale` is valid) after the note is added.

```
s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G'), Note(name='A'), Note(name='B'))

s.remove_note(note=Note(name='B'))
```

(continues on next page)

(continued from previous page)

```

print(len(s.notes))  # 6

for n in s.notes:
    print(n)

# C4
# D4
# E4
# F4
# G4
# A4

```

By default, as before, strict is set to True, so if the Scale will be invalid after remove you will have a ScaleException. If ScaleException is raised the notes in the scale will be restored as they were before the remove.

```

s = Scale(Note(name='C'))

s.remove_note(note=Note(name='C'))
# Invalid Scale.

print(len(s.notes))  # 1

for n in s.notes:
    print(n)

# C4

```

Removing a Note by octave or name can remove multiple notes.

```

s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G
↪'), Note(name='A'), Note(name='B'), Note(name='C', octave=5))
# C4 and C5 are different note

print(len(s.notes))  # 8

s.remove_note(name='C')
print(len(s.notes))  # 6

for n in s.notes:
    print(n)

# D4
# E4
# F4
# G4
# A4
# B4

s = Scale(Note(name='C'), Note(name='D'), Note(name='E'), Note(name='F'), Note(name='G
↪'), Note(name='A'), Note(name='B'), Note(name='C', octave=5), Note(name='C', ↪
↪octave=6))
s.remove_note(octave=5)

print(len(s.notes))  # 7

```

(continues on next page)

(continued from previous page)

```

for n in s.notes:
    print(n)

# C4
# D4
# E4
# F4
# G4
# A4
# B4

```

6.6 is valid

If you need to know if the actual Scale is valid you can use `is_valid` method. A scale is valid if has one or more Note and if all notes are instance of `Note()`

6.7 Comparison

Scale support equal and not equal comparison operator. Check `NoteList` documentation for more information

6.8 Create Scale from root Note

You can easily create a Scale from root note using the `create_from_root` classmethod. Suppose you want create a C major scale.

```

s = Scale.create_from_root(root=Note(name='C'))
for n in s.notes:
    print(n)

# C4
# D4
# E4
# F4
# G4
# A4
# B4

s = Scale.create_from_root(root=Note(name='C'), scale_type=Scale.MINOR_TYPE, alt=Note.
↪FLAT)
for n in s.notes:
    print(n)

# C4
# D4
# Eb4
# F4
# G4
# Ab4
# Bb4

```

That's it, you've got a C major scale and then a C minor scale. `create_from_root` use the classmethod `get_notes_from_root` of `NoteList`. Check `NoteList` documentation for more information.

babs come with some of pre-defined `scale_type` so that the previous example could be the same as

```
s = Scale.create_from_root (root=Note (name='C'), chord_type=Scale.DORIAN_TYPE)
```

You can use a custom list or use some of the pre-defined chord type.

6.9 List of pre-defined scale type

Scale type
MAJOR_TYPE
MINOR_TYPE
IONIAN_TYPE
DORIAN_TYPE
PHRIGIAN_TYPE
LIDYAN_TYPE
DOMINANT_TYPE
AEOLIAN_TYPE
LOCRIAN_TYPE
PENTATONIC_TYPE
PENTATONIC_MINOR_TYPE
BLUES_TYPE
BLUES_MINOR_TYPE
MELODIC_MINOR_TYPE
HARMONIC_MINOR_TYPE
HARMONIC_MAJOR_TYPE

CHAPTER 7

Authors

- Mattia Careddu - <http://github.com/iskyd>

CHAPTER 8

Indices and tables

- `genindex`
- `search`

C

Chord (*built-in class*), 19

create_from_root() (*Chord class method*), 19

create_from_root() (*Scale class method*), 25

G

get_note_index() (*Note method*), 5

get_note_name_by_index() (*Note class method*),
5

N

Note (*built-in class*), 5

P

pitch_shift() (*Note method*), 5

R

Rest (*built-in class*), 11

S

Scale (*built-in class*), 25