# CHAPTER 4

# Rests and Notes as Python Objects

The Python library `pyknon` generates music in a hacker-friendly way. We'll use this library to generate the examples in this book. You can download it at http://kroger.github.com/pyknon.

It's a library intended for teaching and demonstrating, so you should have no problems reading the source code. On the other hand, it doesn't do a lot of checking, so it's easy to break (and if you break it, you buy it). If you find bugs, please report at https://github.com/kroger/pyknon/issues.

We'll use the `music` module inside `pyknon` to describe musical notes and rests as Python objects. It uses Mark Conway Wirt's MIDIUtil library to generate MIDI files (it's included in pyknon). The module `music` has three basic classes, `Note`, `Rest`, and `NoteSeq`. See section *About MIDI* to learn more about the MIDI format.

## 4.1 Rest

`Rest` is a very simple class. It has only one attribute:

`dur`
: the duration value as float point (quarter is 0.25 since 1/4 = 0.25).

And it has only one method:

`stretch_dur(`*factor*`)`
: Multiplies the duration by `factor` and returns a new `Rest` with the resulting duration.

## 4.2 Note

The class `Note` has six attributes and accepts the following attributes as keyword arguments: `value`, `octave`, `dur`, and `volume`.

`value`
: the integer value for a note, from 0 to 11.

`octave`
: the octave value where the central octave is 5.

`midi_number`
: returns the MIDI value for the pitch. That is, `value + (octave * 12)`. This attribute is read-only.

`dur`
: the note value as float point (quarter is 0.25 since 1/4 = 0.25).

`volume`
: MIDI volume value from 0 to 127.

`verbose`
: returns a string in the format `<note>, <octave>, <duration>`.

For example, to instantiate a loud middle C with a duration of a quarter you could write `Note(value=0, octave=5, dur=0.25, volume=127)` or

`Note(0, 5, 0.25, 127)`. `Note` has the following defaults: value=0, octave=5, dur=0.25, volume=100. Therefore, `Note()` will return a middle C with a duration of a quarter and volume of 100db:

```
>>> a = Note()
>>> <C>
>>> a.octave
>>> 5
>>> a.value
>>> 0
>>> a.volume
>>> 100
>>> a.dur
>>> 0.25
>>> a.midi_number
>>> 60
```

You can instantiate a `Note` using a shorthand notation. Besides numbers, you can pass a string as the first argument in the format "`<note name> <duration> <octave>`" (the second two arguments are optional). For instance: `Note("C4'")`.

In this case, the duration is the reciprocal of the note value (that is, 1/4 becomes 4) and octave is either a number of single quotes or commas. There's a catch. If you enter a duration using the string argument you should use the reciprocal value we just saw, but if you use the `dur` attribute directly you should use float points such as 0.25 for quarter notes. The reason to use the reciprocal of the duration is that many music packages such as Humdrum and Lilypond do that. It's a nice shorthand notation.

The number of single quotes indicates the octave with respect to the central octave, where one quote is the central octave, two quotes one octave above the central octave, and so on. The number of commas works in the opposite way: one comma is one octave below the central octave, two commas are two octaves below and so on. It sounds more complicated than it really is.

`Note` prints its value in the format `<note name>`. It's spartan, but it's good when you have a lot of notes (to keep things short). If you want a more descriptive name, you can use the `verbose` property:

```
>>> Note(2)
>>> <D>
>>> Note(”D#”)
>>> <D#>
>>> Note(”Eb8’’”)
>>> <D#>
>>> Note(”F#,”).verbose
>>> <Note: 6, 4, 0.25>
>>> Note(”C#”, dur=2)
>>> <C#>
```

**Exercise 6.** Create some `Note` objects in the Python interpreter. Use both the regular and shorthand notations.

`Note` has four methods, `transposition`, `inversion`, `stretch_dur`, and `harmonize`, but you are more likely to use these methods in a `NoteSeq` than in a single `Note`.

`transposition(`*index*`)`

Transposes a `Note` by a given `index` in semitones and returns a new `Note`.

```
>>> d = Note(”D”)
>>> <D>
>>> d.transposition(3)
>>> <F>
```

`inversion(`*index=0, initial_octave=None*`)`

Inverts a `Note` through an inversion `index` and returns a new `Note`.

```
>>> d = Note(”D”)
>>> <D>
>>> d.inversion()
>>> <A#>
>>> d.inversion(initial_octave=8)
>>> <A#>
```

`stretch_dur(`*factor*`)`

Multiplies the duration by `factor` and returns a new `Note` with the resulting duration.

```
>>> d = Note("D")
>>> <D>
>>> d.dur
>>> 0.25
>>> d1 = d.stretch_dur(2)
>>> <D>
>>> d1.dur
>>> 0.5
>>> d2 = d.stretch_dur(0.5)
>>> <D>
>>> d2.dur
>>> 0.125
```

`harmonize`(*scale*, *interval*, *size*)

Harmonize a single note in the context of a scale. Not very useful by itself, but it's used by `NoteSeq`.

`scale`

A set of notes as a `NoteSeq`.

`interval`

The interval in the scale between each note (for example, 3 for thirds, 4 for fourths, and so on). The default is 3.

`size`

The number of notes in the chord. The default is 3.

## 4.3 NoteSeq

`NoteSeq` is a list-like object that can contain multiple `Note` and `Rest` objects and nothing more. You can use slices and methods like `append` and `insert`, just like a Python list.

You can enter the collection of `Note` and `Rest` objects manually:

```
>>> NoteSeq([Note(0), Rest(1), Note("C#8")])
>>> <Seq: [<C>, <R: 1>, <C#>]>
```

Or you can use a shorthand notation as a string argument where each

Note or Rest is separated by spaces and a Rest is represented by an R. The format string for a individual note is the same shorthand used to instantiate a Note:

```
>>> NoteSeq("C R C#8")
>>> <Seq: [<C>, <R: 0.25>, <C#>]>
```

Here are some more examples:

```
>>> NoteSeq("C#2' D#4''")
>>> <Seq: [<C#>, <D#>]>
>>> NoteSeq("C# D#")
>>> <Seq: [<C#>, <D#>]>
>>> NoteSeq("C8 D E4 F")
>>> <Seq: [<C>, <D>, <E>, <F>]>
>>> NoteSeq([Note(0), Note(2)])
>>> <Seq: [<C>, <D>]>
>>> NoteSeq([Note(0, 5), Note(2, 5)])
>>> <Seq: [<C>, <D>]>
>>> NoteSeq([Note(1, 5, 2), Note(3, 6, 1)])
>>> <Seq: [<C#>, <D#>]>
>>> NoteSeq([Note(1, dur=0.5), Note(3, dur=1)])
>>> <Seq: [<C#>, <D#>]>
```

The default value for duration is 4 (quarter note) and for octave is ' (single quote, central octave). If the duration and/or octave of a note is not defined, it will use the value of the previous note. Here's a quick way to have a bunch of eighth notes:

```
>>> a = NoteSeq("C8 D E F")
>>> <Seq: [<C>, <D>, <E>, <F>]>
>>> [x.dur for x in a]
>>> [0.125, 0.125, 0.125, 0.125]
```

It seems laborious to use the Note and Rest objects in a NoteSeq when you have the string shorthand, but it can be particularly useful when generating a NoteSeq from Python code. In the following example we generate a NoteSeq with the whole-tone scale:

```
>>> NoteSeq([Note(x) for x in range(0, 12, 2)])
>>> <Seq: [<C>, <D>, <E>, <F#>, <G#>, <A#>]>
```

Finally, you can instantiate a `NoteSeq` by passing a filename as an argument in the format `file://<filename>`. The content of the filename should be notes and rests separated by spaces just like the string shorthand. Newline characters don't count, and you can use them to make the file more readable. Let's say we have a file called "notes" with the following content:

```
C4 D8 E
F2
G4 A4
```

We can read it with the following code:

```
>>> NoteSeq("file://code-example/notes")
>>> <Seq: [<C>, <D>, <E>, <F>, <G>, <A>]>
```

All the music operations we defined in section *Some Music Operations* are now methods in `NoteSeq`. For example, to calculate the inversion of a sequence:

```
>>> seq = NoteSeq("C E G")
>>> <Seq: [<C>, <E>, <G>]>
>>> seq.inversion()
>>> <Seq: [<C>, <G#>, <F>]>
```

Like a regular Python list, you can concatenate multiple `NoteSeq` using the + operator:

```
>>> NoteSeq("C D E") + NoteSeq([Note(5)])
>>> <Seq: [<C>, <D>, <E>, <F>]>
```

And you can repeat a `NoteSeq` by multiplying it by a number:

```
>>> NoteSeq("C D E") * 3
>>> <Seq: [<C>, <D>, <E>, <C>, <D>, <E>, <C>, <D>, <E>]>
```

These are the main methods in `NoteSeq`:

`retrograde()`
: Returns a new `NoteSeq` with the order of items reversed.

`transposition(`*index*`)`
: Returns a new `NoteSeq` with the notes transposed by a transposi-

tion index, in which the index is an integer. It'll transpose only the notes and leave the rests untouched.

```
>>> a = NoteSeq("C4 D8 R E")
>>> <Seq: [<C>, <D>, <R: 0.125>, <E>]>
>>> a.transposition(3)
>>> <Seq: [<D#>, <F>, <R: 0.125>, <G>]>
```

`transposition_startswith(`*note_start*`)`

Transpose a `NoteSeq` in a way that the transposed sequence will start with `note_start`. The argument `note_start` can be a `Note` or an integer representing a pitch from 0 to 11.

```
>>> a = NoteSeq("C4 D8 R E")
>>> <Seq: [<C>, <D>, <R: 0.125>, <E>]>
>>> a.transposition_startswith(Note(3))
>>> <Seq: [<D#>, <F>, <R: 0.125>, <G>]>
>>> a.transposition_startswith(3)
>>> <Seq: [<D#>, <F>, <R: 0.125>, <G>]>
```

`inversion(`*index=0*`)`

Returns a new `NoteSeq` with the notes inverted by an inversion index. I think the method `inversion_startswith` is more useful and easier to use.

`inversion_startswith(`*note_start*`)`

Inverts a `NoteSeq` in a way that the inverted sequence will start with `note_start`. The argument `note_start` can be a `Note` or an integer representing a pitch from 0 to 11.

`rotate(`*n=1*`)`

Returns a new `NoteSeq` with the items rotated by *n*.

`stretch_dur(`*factor*`)`

Returns a new `NoteSeq` with the duration of each item multiplied by `factor`. It's good for making a sequence of notes longer or shorter.

`stretch_inverval(`*factor*`)`

Returns a new `NoteSeq` with the intervals stretched by `factor`. This is useful when adding variation to a set of notes while keeping the same contour.

```
>>> a = NoteSeq("C D E")
>>> <Seq: [<C>, <D>, <E>]>
>>> a.stretch_inverval(2)
>>> <Seq: [<C>, <E>, <G#>]>
```

`harmonize`(*interval*, *size*)

Returns all harmonizations for the `NoteSeq` as a new `NoteSeq`.

`interval`

The interval in the scale between each note (for example, 3 for thirds, 4 for fourths, and so on). The default is 3.

`size`

The number of notes in the chord. The default is 3.

## 4.4 Generating MIDI files

To generate a MIDI file we use the `Midi` class in `pyknon.genmidi`. The following example shows the basic usage (I'm using `from __future__ import division` so I can write durations as 1/4 instead of 0.25).

```
def demo():
    notes1 = NoteSeq("D4 F#8 A Bb4")
    notes2 = NoteSeq([Note(2, dur=1/4), Note(6, dur=1/8),
                      Note(9, dur=1/8), Note(10, dur=1/4)])
    midi = Midi(number_tracks=2, tempo=90)
    midi.seq_notes(notes1, track=0)
    midi.seq_notes(notes2, track=1)
    midi.write("midi/demo.mid")
```

We define a `Midi` object with two tracks and a tempo of 90 beats per second and write the `NoteSeq` in `notes1` to the first track, and the `NoteSeq` in `notes2` to the second track. Finally, we write the whole thing to the file "demo.mid".

The class `Midi` has the following attributes:

`number_tracks`

The number of MIDI tracks. The default is 1.

`tempo`
: The MIDI tempo, from 0 to 127 BPM. The default is 60.

`instrument`
: The MIDI instrument value from 0 to 127. The default is 0 (piano).

It has the following methods:

`seq_notes(`*note_seq*, *track=0*, *time=0*`)`
: Writes a `NoteSeq` to the MIDI stream.

  `note_seq`
  : A sequence of notes of type `NoteSeq`.

  `track`
  : Specifies the track number in which the note sequence will be appended. Default is 0.

  `time`
  : The number of beats the note sequence will start. Default is 0.

`write(`*filename*`)`
: Writes the MIDI stream to `filename`.

**Exercise 7.** In the following code, what is the order of the notes in the MIDI file? What happens when you change the second-to-last line to `midi.seq_notes(seq2, time=3)` or `midi.seq_notes(seq2, time=4)`?:

```python
from pyknon.genmidi import Midi
from pyknon.music import NoteSeq

seq1 = NoteSeq("C D E")
seq2 = NoteSeq("F G A")

midi = Midi()
midi.seq_notes(seq1)
midi.seq_notes(seq2)
midi.write("foo.mid")
```

## 4.5 About MIDI

The Musical Instrument Digital Interface (MIDI) specification was developed in the 1980s to exchange information between keyboard synthesizers. The MIDI file format is low-level and it doesn't have the notion of notes, rests, and duration values such as quarter notes. It's built around messages such as `Note On` and `Note Off`.

A MIDI file holds information about when a note started and stopped, but it doesn't know how the music will actually sound in the way a MP3 file knows. A MIDI player will either synthesize the sound or use sound samples (the popular SoundFont format uses sound samples). This means that a MIDI file may sound wonderful in one program or computer and appaling in others.

One advantage is that MIDI files are easy to generate and have a small size. For instance, the size of a MIDI file containing the first movement of Beethoven's Ninth Symphony is less than 220k (we're talking about tousands of notes here) while an MP3 of a recording of the same Symphony is almost 28Mb. Just keep in mind that the enjoyment factor while listening is proportional to the file size.