

Music for Geeks & Nerds

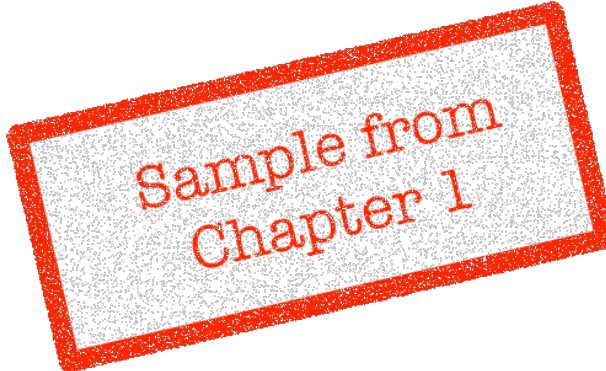
**Learn more about music with
Python and a little bit of math**

Pedro Kroger

Contents

1	Introduction	1
1.1	Getting Started	2
1.2	Acknowledgments	3
2	Introduction to Music Notation in n Seconds or Less	4
3	The Primitives of Music	8
3.1	Notes	8
3.2	Accidentals	11
3.3	Integer Notation	13
3.4	Octaves	16
3.5	Note Value	16
3.6	Some Music Operations	20
4	Rests and Notes as Python Objects	31
4.1	Rest	32
4.2	Note	32
4.3	NoteSeq	35
4.4	Generating MIDI files	39
4.5	About MIDI	40
5	Means of Combination	42
5.1	Random Combination	42

5.2	Music from Math	45
5.3	Combination of Notes Horizontally	47
5.4	Chords: Combination of Notes Vertically	52
5.5	Summary	58
6	A Look Inside the Primitives	59
6.1	The Basics of Sound	59
6.2	The Harmonic Series: a Building Block	62
6.3	The Beautiful Math of Temperament Systems	65
7	Means of Abstraction	69
7.1	Example: Sergei Rachmaninoff, <i>Vocalize</i>	72
7.2	Conclusion	73
8	Conclusion and Next Steps	74
9	List of Exercises	76
10	List of Tracks	78



Sample from
Chapter 1

CHAPTER 1

Introduction

I have a lot of friends who are computer scientists and engineers, and they are always asking me for books to learn more about music. Unfortunately, I have never found a good book to recommend. There are good books out there, but they present things magically instead of logically and tend to be kind of patronizing. Because music is an ancient art with more than 2500 years of recorded data, it has a lot of baggage. Things like tetrachords that the Greeks used more than 2500 years ago are still used today. This is good, but sometimes it is difficult to separate what is natural (such as frequencies); logical (such as the math used for transposition and inversion); from what is the result of social conventions and usage over hundreds of years (such as interval names). In this book, I clearly separate these three things. If you are a nerd like me, you will find that you can learn the natural and logical stuff pretty quickly.

I am heavily influenced by Hal Abelson and Jerry Sussman's *Structure and Interpretation of Computer Programs* (<http://bit.ly/sicp-book>), in which they state that every powerful language has three mechanisms for combining simple elements to form more complex ideas:

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

In the same way that we can apply these mechanisms to programming, I like to apply them to music. In fact, I use this idea in teaching both computer science and music composition, and I find it very powerful. We'll explore some of these ideas in this book.

Another similar font of inspiration is the talk “Why Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas” by Gerald J. Sussman (<http://bit.ly/why-programming>) in which he uses programming to teach electrical engineering. In this book we will use programming to learn music.

We will see some music notation in this book, but don't worry. If you have never seen music notation before, read chapter *Introduction to Music Notation in n Seconds or Less*. If you don't feel like learning music notation at all, it's no problem; you should be all right by reading the text and code examples.

1.1 Getting Started

This book has quite a few code and audio examples. You can download them at our [resources webpage](#).

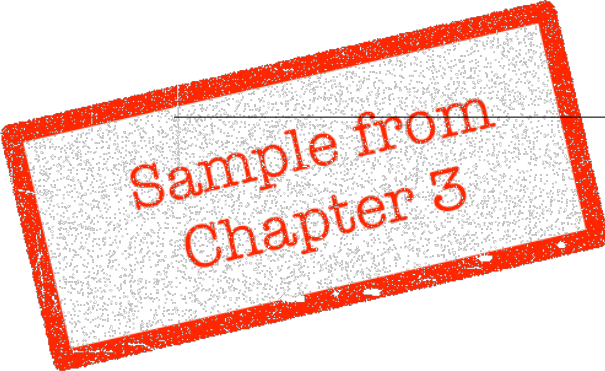
The sound examples in this book are in both MIDI and MP3 formats. I used a high quality sampler library to generate the MP3s from MIDI, so it should sound good.

To play the MIDI files you'll create using the `genmidi` module, you'll need a MIDI player. Windows and Ubuntu should play MIDI files by default when you double-click on them. On a Mac you may need to install QuickTime Player 7. Another option is to use a music notation

program. A music notation program may be useful even if you don't know how to read music, as it'll help you spot crazy outputs. [Musescor](#)e is a free program (as in speech and beer) that runs on Linux, Mac, and Windows. [Finale](#) is a commercial notation program for Windows and Mac that is more polished than Musescor, but it's expensive. Their trial version is fully functional and works for 30 days.

1.2 Acknowledgments

This book would not be possible without the encouragement of many people. I'd like to thank the nice folks who took my tutorial at the 2012 PyCon. The tutorial was based on an earlier draft of this book, and their participation was essential in cleaning the material and inspiring me to finish this book. I'd like to thank Vilson Vieira, Marcos Sampaio, Alexandre Passos, and Tiago Vaz for their invaluable suggestions. Finally, I'd like to acknowledge Mara for her patience, love, and support.



Sample from
Chapter 3

To calculate the duration in seconds of a note we use the formula $60n/vt$, where n is the note value, v is the note value of the tempo (“quarter”, as in $\text{♩} = 90$), and t is the tempo itself.

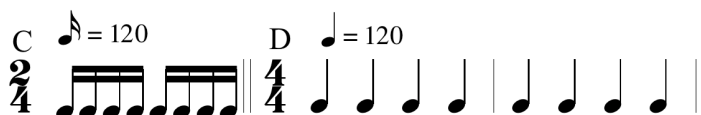
The function `note_duration` returns the time in seconds a note value has in a specific tempo:

```
def note_duration(note_value, unity, tempo):
    return (60.0 * note_value) / (tempo * unity)
```

We can see the result for the examples we mentioned before:

```
>>> note_duration(1/4, 1/4, 90)
>>> 0.6666666666666667
>>> note_duration(1/2, 1/4, 90)
>>> 1.3333333333333333
>>> note_duration(1/8, 1/4, 90)
>>> 0.3333333333333333
```

I hope it’s easy to see that in the following image the bars marked with C and D will sound exactly the same, even if they use different note values (sixteenth and quarter notes, respectively):



The utility function `durations` returns the duration in seconds of a list of note values:

```
def durations(notes_values, unity, tempo):
    return [note_duration(nv, unity, tempo) for nv in notes_values]

>>> durations([1/2, 1/4, 1/8], 1/4, 60)
>>> [2.0, 1.0, 0.5]
>>> durations([1/2, 1/4, 1/8], 1/4, 120)
```

```
>>> [1.0, 0.5, 0.25]
>>> durations([1/2, 1/4, 1/8], 1/4, 90)
>>> [1.3333333333333333, 0.6666666666666666, 0.3333333333333333]
```

Sometimes a note value can have a dot that will increase the total duration by half of the original note value. For instance, ♩ has the value of 1/4, while ♩. has the value of 1/4 + 1/8 = 3/8. Each dot increases the total value by half of the previous value: ♩.. = 1/4 + 1/8 + 1/16 = 7/16. To get the total value we can use the formula for the sum of a geometric series:

$$\sum_{k=0}^{n-1} ar^k = a \frac{1 - r^n}{1 - r}$$

Where a is the first term of the series and r is the common ratio. Having the formula, the implementation in code is straightforward (I'm using Python's class `Fraction` from the module `fractions`):

```
def dotted_duration(duration, dots):
    ratio = Fraction(1, 2)
    return duration * (1 - ratio ** (dots + 1)) / ratio
```

```
>>> dotted_duration(Fraction(1,4), 0)
>>> 1/4
>>> dotted_duration(Fraction(1,4), 1)
>>> 3/8
>>> dotted_duration(Fraction(1,4), 2)
>>> 7/16
```

Exercise 3. Create a function `music_duration` to calculate the total duration in minutes of a composition. This function accepts four parameters: the time signature as a string, the number of bars, the reciprocal of the note value of the tempo (for example, use 4 if the note value is a quarter), and the tempo itself. It assumes that the tempo and time signature of a composition won't change. To find the duration of 10 bars in a composition that has a time signature of "4/4" and a tempo marking of ♩ = 60, you'd have the following function call: `music_duration("4/4", 10, 4, 60)`.

3.6 Some Music Operations

In this section we are going to see a simple library to represent music notes (the module `simplemusic` in `pyknon`). In chapter *Rests and Notes as Python Objects* we'll use `music`, an object-oriented module that is more complete. In `simplemusic` notes are represented as numbers, and a set of notes is represented as a Python iterable (usually a list).

3.6.1 Intervals

Mathematically, a musical interval is the difference in semitones between two notes. Intervals can also be represented with integers:

```
def interval(x, y):
    return mod12(x - y)
```

We can see that the interval from D and E is 2 semitones and from E to D is 10 semitones:

```
>>> interval(2, 4)
>>> 10
>>> interval(4, 2)
>>> 2
```

The reason that the same notes in different order will have different intervals has to do with direction. There are 2 semitones from D to the next E:

C C# D D# E F F# G G# A A# B
|-----2----->

If we follow the same direction, there are 10 semitones from E to the next D:

C C# D D# E F F# G G# A A# B C C# D
|-----10----->

If we change the direction we need to change the signal, so we have -2 semitones from E to the *previous* D:

C C# D D# E F F# G G# A A# B
|-----2-----|

And, of course, $-2 \bmod 12$ is 10.

These two intervals (from D to E and from E to D) are complements and their sum is always 12. For instance:

```
>>> interval(3, 7)
>>> 8
>>> interval(7, 3)
>>> 4
```

See section *Interval Names* for an Python implementation of interval names such as “minor third” and “perfect fifth”.

3.6.2 Transposition

Music transposition is an operation that shifts a set of notes up or down by a constant interval. Mathematically, transposition is the sum of each note in a group of notes by a transposition index:

```
def transposition(notes, index):
    return [mod12(n + index) for n in notes]

>>> scale = [0, 2, 4, 6, 8, 10]
>>> [0, 2, 4, 6, 8, 10]
>>> notes_names(scale)
>>> ['C', 'D', 'E', 'F#', 'G#', 'A#']
>>> transposition(scale, 3)
>>> [3, 5, 7, 9, 11, 1]
>>> notes_names(transposition(scale, 3))
>>> ['D#', 'F', 'G', 'A', 'B', 'C#']
```

3.6.3 Retrograde

Retrograde is the reverse of a group of notes and it's straightforward to implement in Python:

```
def retrograde(notes):
    return list(reversed(notes))
```

```
>>> scale = [0, 2, 4, 6, 8, 10]
>>> [0, 2, 4, 6, 8, 10]
>>> retrograde(scale)
>>> [10, 8, 6, 4, 2, 0]
```

3.6.4 Rotation

Rotation is an useful operation in music. The following function accepts an iterable and a rotation index as arguments:

```
def rotate(item, n=1):
    modn = n % len(item)
    return item[modn:] + item[0:modn]

>>> scale = [0, 2, 4, 6, 8, 10]
>>> [0, 2, 4, 6, 8, 10]
>>> rotate(scale, 3)
>>> [6, 8, 10, 0, 2, 4]
```

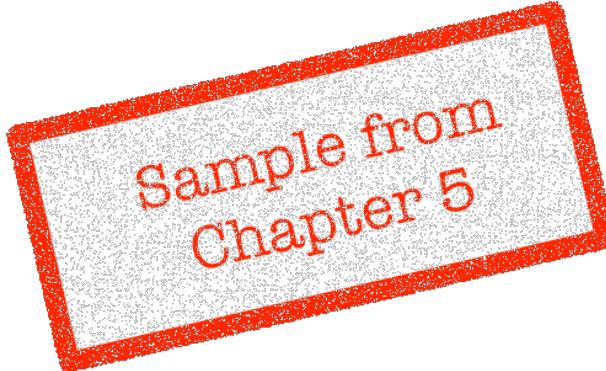
We can see rotation in action in medieval modes where each mode is a rotation of the other:

```
>>> dorian = "D E F G A B C".split()
>>> ['D', 'E', 'F', 'G', 'A', 'B', 'C']
>>> phrygian = rotate(dorian)
>>> ['E', 'F', 'G', 'A', 'B', 'C', 'D']
>>> lydian = rotate(phrygian)
>>> ['F', 'G', 'A', 'B', 'C', 'D', 'E']
>>> mixolydian = rotate(lydian)
>>> ['G', 'A', 'B', 'C', 'D', 'E', 'F']
```

3.6.5 Inversion

Inversion is used a lot in music, and, as many things in music, it has different meanings.

Chord inversion is actually a rotation where the lowest note of a chord changes according to the rotation:



Sample from
Chapter 5

CHAPTER 5

Means of Combination

We can combine the music primitives to form more complex entities. This, along with transformation using music operations, is in the heart of music composition and has been used for hundred years.

5.1 Random Combination

Before we combine the music primitives we have seen in chapter *The Primitives of Music*, let's generate some music randomly to have an idea of how it sounds.

The functions discussed in this section are in the file `random_combination.py`. This file has a few utilities functions such as `choice_if_list` and `genmidi` that we won't see here since they are simple and boring. You may explore them in the source file.

The function `random_notes` generates a sequence of notes by choosing a pitch randomly from a list of pitches. The second, third, and fifth arguments define the octave, duration, and volume, respectively. If any of

these arguments is a list, `choice_if_list` will pick one element randomly or return the argument itself if it's a number. Finally, the argument `number_of_notes` contains how many notes we want to generate.

```
def random_notes(pitch_list, octave, duration,
                 number_of_notes, volume=120):
    result = NoteSeq()
    for x in range(0, number_of_notes):
        pitch = choice(pitch_list)
        octave = choice_if_list(octave)
        dur = choice_if_list(duration)
        vol = choice_if_list(volume)
        result.append(Note(pitch, octave, dur, vol))
    return result
```

In the following example we want to generate five notes from the chromatic scale, in any octave from five to six, with quarter note, eighth note, or sixteenth note durations:

```
>>> random_notes(range(0, 12), range(5, 7), [0.25, 0.5, 1], 5)
>>> <Seq: [<D#>, <F#>, <G>, <C>, <A>]>
```

In the following example we generate random notes from the pentatonic scale, in the central octave, with a duration of an eighth note:

```
>>> random_notes([0, 2, 4, 7, 9], 5, 0.5, 5)
>>> <Seq: [<C>, <G>, <D>, <G>, <A>]>
```

Let's generate a hundred notes from the chromatic scale, in any octave from 0 to 8 (that's quite a range!), and using all basic durations. (We're using from `__future__` import `division`, so we can type things like `1/4` instead of `0.25`).

```
def random1():
    chromatic = range(0, 12)
    durations = [1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 1]
    notes1 = random_notes(chromatic,
                          range(0, 9),
                          durations,
                          100,
                          range(0, 128, 20))
```

```
gen_midi("random1.mid", notes1)
```

Track 2. Notice how this track sounds. Do you think it sounds similar to the music you enjoy? There's no right answer here, but most people will think this doesn't sound good. Even if you find it interesting at first, it may get boring after a while (try it with a thousand notes!). But I don't want to control your listening here; if you like it, we'll still love you.

Now let's add some restrictions. We'll generate the same hundred random notes from the chromatic scale, but this time with a smaller range and with only two durations:

```
def random2():
    chromatic = range(0, 12)
    notes2 = random_notes(chromatic,
                          range(3, 7),
                          [1/16, 1/8],
                          100)
    gen_midi("random2.mid", notes2)
```

Track 3. What do you think? I imagine you'll agree that it sounds much more familiar than the previous track. This is because we are using a more restricted octave range and note values.

Now we'll generate another hundred notes from the pentatonic scale, in any octave from 5 to 6 (only two octaves), and with a duration of a sixteenth note:

```
def random3():
    pentatonic = [0, 2, 4, 7, 9]
    notes = random_notes(pentatonic,
                        range(5, 7),
                        1/16,
                        100)
    gen_midi("random3.mid", notes)
```

Track 4. By having only one note value and a very recognizable scale, this example may sound the most familiar to you.

Naturally, nobody can tell how you should listen to things. Maybe `random1` was the example you liked the best. However, one important point

here is that random music almost never sounds good or familiar. Repetition and constraints are important.

Exercise 8. Generate some random notes using `random1`, `random2`, and `random3` but using the major and minor scales.

Exercise 9. Play with `random_notes` to generate different notes. Add different kinds of constraints and see which ones you like the best.

5.2 Music from Math

Often, beautiful mathematics doesn't make beautiful music and vice versa, but sometimes it does.

Let's define a function `play_list` that is similar to `random_notes`, but instead of picking random notes from a list, it receives a list of numbers and turn them into notes:

```
def play_list(pitch_list, octave_list, duration,
              volume=120):
    result = NoteSeq()
    for pitch in pitch_list:
        note = pitch % 12
        octave = choice_if_list(octave_list)
        dur = choice_if_list(duration)
        vol = choice_if_list(volume)
        result.append(Note(note, octave, dur, vol))
    return result
```

Now let's see how Fibonacci's sequence and Pascal's triangle sound. Pascal's triangle is an array of the binomial coefficients that has all kinds of neat properties. Check [Pascal's Triangle And Its Patterns](#) for more. Here are the first six rows:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Did you enjoy this sample?

please visit

<http://musicforgeeksandnerds.com>

to purchase your copy.