Music for Geeks & Nerds

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

Pedro Kroger

Contents

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. Musescore 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 Musescore, but it's expensive. Their trial version is fully functional and works for 30 days.

1.2 Acknowledgments

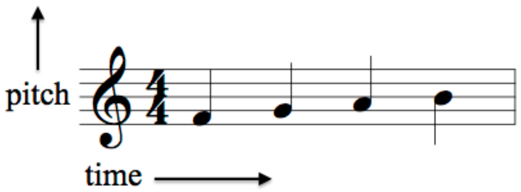
This book would not be possible without the encouragement of many people. I'd like to thank the nice folks who took my tutorial at the 2012 PyCon. The tutorial was based on an earlier draft of this book, and their participation was essential in cleaning the material and inspiring me to finish this book. 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.

CHAPTER 2

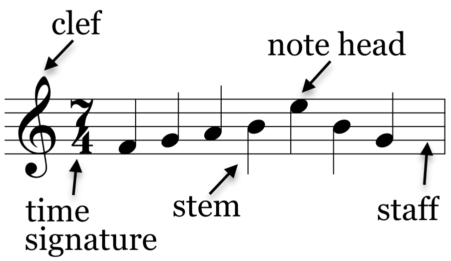
Introduction to Music Notation in *n* Seconds or Less

Music notation is both simple and complex, not unlike math notation. In this chapter we'll learn enough music notation to understand the examples in this book.

The first thing to know about music notation is that time is represented from left to right while pitch is represented from bottom to top:



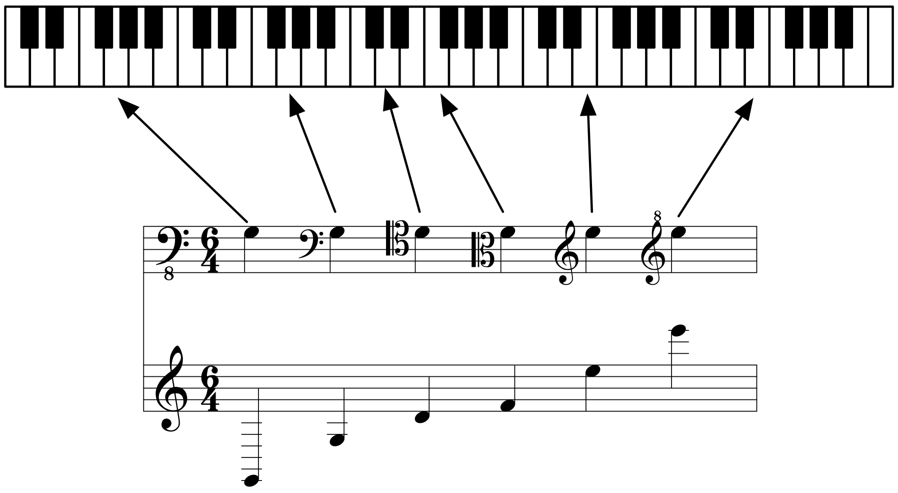
As you'd expect, there are symbols to represent different things. The main symbols are the clef, the time signature, the note head, and the stem, as you can see in the following image:



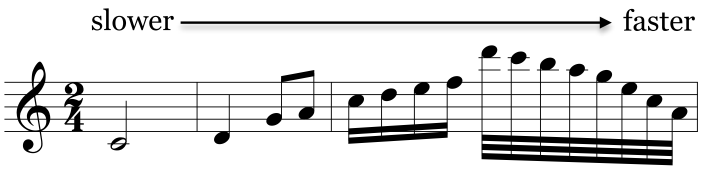
The time signature shows how many beats are in a measure and the note value of the beat. For instance, in the previous image we have seven beats (represented by the number 7) of quarter notes (represented by the number 4). Naturally, we can subdivide a beat into smaller note values (see section *Note Value*).

The actual pitch of a note is determined by the position of a note head on the staff and the clef being used. Clefs solve the problem of cramming 80+ notes into five lines. There are three kinds of clefs—G-clef, F-clef, and C-clef—and they indicate the pitch name on a specific line. For instance, the treble clef (the one used in the two previous images) indicates that the second line from the bottom has the note G, while the bass clef indicates that the fourth line from the bottom has the low F note.

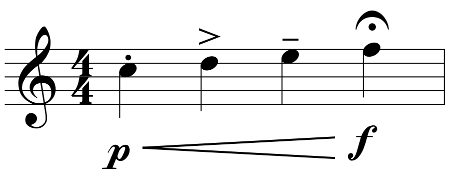
In the following image we can see six notes in the same graphical position in the top staff. Because each note occurs after a different clef, they will represent different pitches. The arrows show the position of each note on the piano, while the bottom staff shows where these notes would be placed in the treble clef. The sub- and superscript eight indicates that a note will sound one octave below or above than written, respectively.



The note shape and the number of beams (the horizontal lines connecting two or more notes) indicate the note duration. In the following example the notes have shorter duration as the music progresses (see section *Note Value*). Unlike Python, but like the C programming language, space is not significant, but it is used to make a score easy to read. Notice how the space between the first and second note is greater than the space between the third and fourth note. This is because the first note is longer than the third:



Finally, you may see symbols above or below a note such as a dot, a greater than (>), a line, or a fermata (on the last note in the following image). They are symbols of *expression* to tell the performer how the note should be played. There are dozens of expression symbols, but we don't need to worry about them in this book:



In this chapter we had the shortest introduction to music notation in the history of introductions, but this is pretty much all you need to know to follow the music examples in this book.

CHAPTER 3

The Primitives of Music

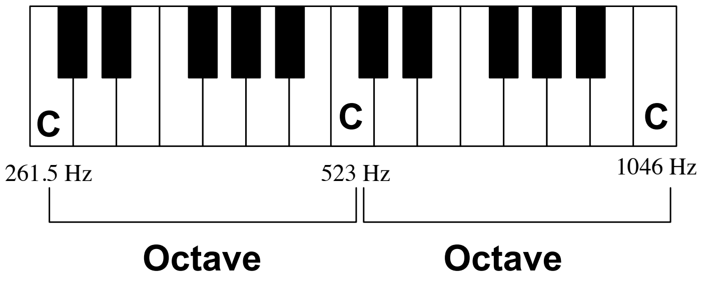
In this chapter we'll see an overview of notes, intervals, and durations and how to represent them in Python.

3.1 Notes

A note is a symbol representing a musical sound.

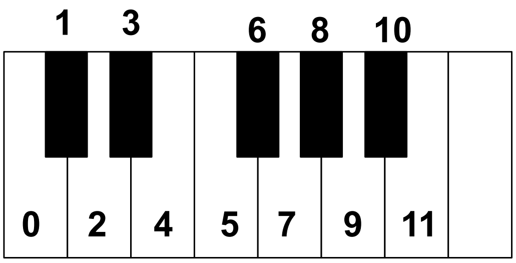
Pitch is the combination of a frequency in Hertz and a note name. For instance, central A has a frequency of 440Hz. This combination is somewhat arbitrary. In the past, the frequency of central A would change depending on the region. In some places, it was equal to 315Hz, for example.

An octave is the interval between two pitches where their frequency has a ratio of 2:1. For example, central A has a frequency of 440Hz and the A one octave above has a frequency of 880Hz. "Two notes an octave apart are in a sense alike, being different only in their relative registers and often seeming to blend into one another" *(Drabkin, 2012)*. In most cultures the two frequencies are perceived as the same pitch in different registers. In the following image we have two octaves. Notice how the pattern of black keys is repeated in each octave:



There are many ways to divide an octave. In fact, octave division has been an important component in music theory for 2500+ years. Today, the most common way to divide the octave is into 12 equal parts, the so-called equal temperament. The equal temperament became prevalent in the 18th century, and before that many others temperaments were used (see section *The Beautiful Math of Temperament Systems*)

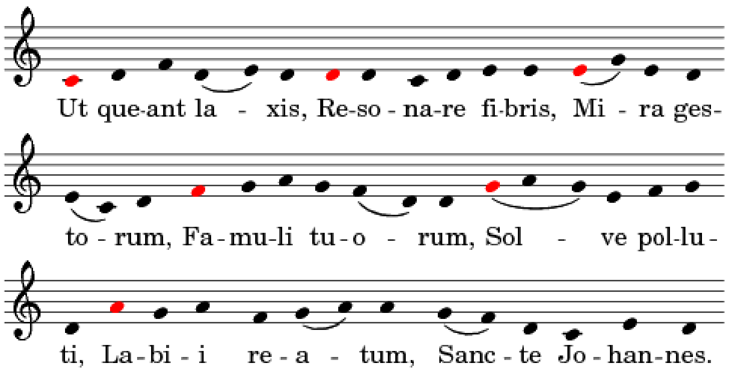
We could name these 12 notes any way we wanted. The geek in us would like to have an array of 12 notes and access them with something like notes[0], notes[1], and so on until notes[11] (as we can see in the following image). In fact, we'll do something similar with integer notation. (see section *Integer Notation*).



The ancient Greeks had many ways of dividing the octave. A specific division was called *genos*, and the plural *genera*. My favorite music-related quote of all time is from Cleonides (1st century BCE): "In respect to pitch, notes are infinite; in respect to function, there are eighteen in each genos" *(Cleonides, 1998)*. The Greeks had a system with 18 notes, and they used names for notes like *proslambanomenos* ("added note") and *lichanos* ("licking finger," that is, forefinger). In this system some notes are movable (that is, their intonation can change) while others are immovable. In the following figure we can see three *genera*: enharmonic, chromatic, and diatonic. The immovable notes are in bold face and the symbol ↑ is used for the enharmonic diesis, an interval with a ratio of 128/125 (see chapter *A Look Inside the Primitives*):

		enhar.	chr.	diat.
	Proslambanomenos	A	A	A
Hypaton	**Hypate**	B	B	B
	Parhypate	B↑	c	c
	Lichanos	c	c♯	d
	Hypate	e	e	e
Meson	Parhypate	e↑	f	f
	Lichanos	f	f♯	g
	Mese	a	a	a
Diezeugmenon	**Paramese**	b	b	b
	Trite	b↑	c'	c'
	Paranete	c'	c♯'	d'
	Nete	e'	e'	e'
Hyperbolaion	Trite	e↑'	f'	f'
	Paranete	f'	f♯'	g'
	Nete	a'	a'	a'

The use of the letters A to G to name notes is attributed to Boethius in the sixth century. The use of the syllables *ut*, *re*, *mi*, *fa*, *sol*, *la*, and *si* is attributed to Guido d'Arezzo as a mnemonic device using the first letters of the hymn *Ut queant laxis* in which every phrase begins on a successively higher note (from C to A, see the following picture). In 1600 Giovanni Battista Doni suggested to replace *ut* with *do* to make it easier to sing, and to add *si* (from the initials for *Sancte Johannes*) to complete the diatonic scale. However, other syllables have been proposed in the late 1500s, such as *be*, *ce*, *di*, *ga*, *la*, *mi*, and *ni* by Waelrant and *la*, *be*, *ce*, *de*, *me*, *fe*, and *ge* by Hitzler *(McNaught, 1893)*. I'm glad these two systems didn't prevail or we would be singing in baby talk today.

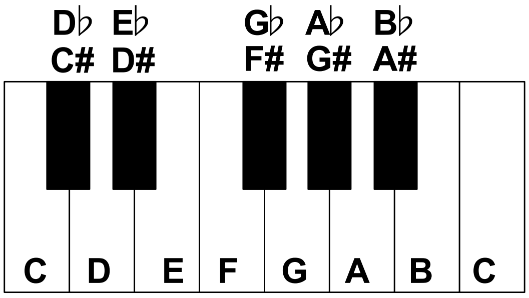

Ut que-ant la - xis, Re-so - na-re fi-bris, Mi - ra ges-
to - rum, Fa-mu-li tu-o - rum, Sol - ve pol-lu-
ti, La-bi - i re - a - tum, Sanc - te Jo - han-nes.

If these last paragraphs look dense it is because I'm trying to cram 2000+ years of history into a couple paragraphs (silly me). But the main point here is that there's nothing magical about the way notes are named; it's more a historical accident than anything else. Speaking of accidents, we should segue to accidentals.

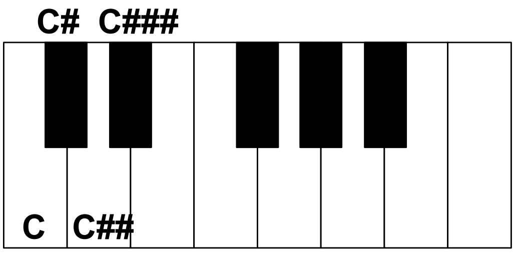
3.2 Accidentals

You must be asking, if we divide the octave in twelve parts and they used seven letters (A to G) or names (ut to si), how do we name the other five notes?

We take one of the seven note names, such as C, and add an accidental; a sign that raises or lowers a pitch one or more semitones. A semitone is the smallest interval used in Western tonal music. The flat (b) lowers a semitone and the sharp (#) raises a semitone. A piano keyboard is a good way to visualize the notes. On the white keys you have the seven notes, from A to G. On the black keys you have the five notes with accidentals:



I hope the picture below helps you understand how accidentals work. C# (note 2) is a semitone higher than C (note 1), C## (C double sharp, note 3) is two semitones higher than C, and C### (C triple-sharp, note 4) is three semitones higher than C. Likewise, Bb is one semitone lower than B, Bbb (B double-flat) is two semitones lower than B, and so on (see section *Integer Notation* for a Python implementation).



In theory you can have as many accidentals you want, but in practice we use at most two. There are a few examples of triple-sharps and triple-flats in the literature, but they are quite rare (You can find them in the works of Charles-Valentin Alkan and Max Reger).

If you think this is more complicated than it should be, it is! It'd be much easier if each of the 12 notes had a different name. But as with many things in music, the way we name notes evolved from the medieval period when they used six notes (hexachords) and sets with seven notes (scales), and deviations from these systems where marked as "accidentals." The other five notes were added gradually to "correct" dissonant intervals such as the tritone from F to B (if you change B to Bb you have a perfect fourth). See sections *Intervals* and *Interval Names* to learn more about intervals and to see how we can implement them in Python.

Also remember that they didn't have a full vision of a 12 notes system. Hell, they didn't even have keyboards! Their system was more complex, but in many ways more colorful, since each scale had its own intonation. To go into more detail is out of the scope of this book, but Wikipedia's entry on musical mode is pretty good.

To summarize, there are many ways to name the notes. On a modern, 12-tone equal temperament we use these names:

n	Name	Other Names
0	C	B#, Dbb
1	C#, Db	
2	D	Ebb
3	D#, Eb	
4	E	Fb
5	F	E#, Gbb
6	F#, Gb	
7	G	Abb
8	G#, Ab	
9	A	
10	A#, Bb	Cbb
11	B	Cb

3.3 Integer Notation

It's common to use integers to represent pitches, and it works really well with equal temperament. Every note in the octave is represented by an integer from 0 to 11 (C to B). This system wraps around 12, just like a 24-hour clock. For instance, 14 and 2 represent the note D, just like 14:00 is the same as 2 p.m.

It's straightforward to implement a `mod12` function with Python:

```
def mod12(n):
    return n % 12
```

And it's equally easy to implement a simple function to show a note name given an integer:

```
def note_name(number):
    notes = ”C C# D D# E F F# G G# A A# B”.split()
    return notes[mod12(number)]
```

```
>>> note_name(0)
>>> C
>>> note_name(1)
>>> C#
>>> note_name(13)
>>> C#
>>> note_name(3)
>>> D#
```

In this function we are limited to showing only sharps (and we could change it to show only flats). To have a more intelligent program capable of distinguishing sharps and flats we need a more elaborate pitch encoding than this base-12 system. For an example of such encoding, see "A Base-40 Number-line Representation of Musical Pitch Notation" by Walter B. Hewlett.

As we saw, a sharp raises a note by a semitone while a flat lowers it by the same amount. If C = 0, then C# = 1, C## = 2, C### = 3, and so on.

A quick way to calculate the number that represents a note (for instance, C###) is to separate the note name from the accidentals (C and ### in our example), get the integer that represents the note name without the accidentals (C = 0), count the number of accidentals, add a plus signal if the accidentals are sharps or a minus if they are flats, and sum the note number with the accidentals. You can see a naive implementation below:

```
def accidentals(note_string):
    acc = len(note_string[1:])
    if "#" in note_string:
        return acc
    elif "b" in note_string:
        return -acc
    else:
        return 0

def name_to_number(note_string):
    notes = "C . D . E F . G . A . B".split()
    name = note_string[0:1].upper()
    number = notes.index(name)
    acc = accidentals(note_string)
    return mod12(number + acc)
```

This is how `name_to_number` works:

```
>>> name_to_number("C#")
>>> 1
>>> name_to_number("Db")
>>> 1
>>> name_to_number("Ebb")
>>> 2
>>> name_to_number("B#")
>>> 0
```

This function, of course, won't work with crazy inputs such as "C#b#", but most (sane) musicians will argue that notes like "C#b#" "don't exist." The hacker in you may want to define a function where this works, as an exercise. What is the integer representation of C#b#? I think it's fair to say it should be 1 (the same as C#), since the flat will cancel one of the sharps.

Exercise 1. Extend the function `name_to_number` to deal with notes with mixed flats and sharps

Exercise 2. Change the function `name_to_number` to use Hewlett's base-40 system.

So, why this funny business about having an arbitrary number of accidentals? The short answer is, except in equal temperament, notes like C# and Db are different, that is, they have different frequencies (see section *The Beautiful Math of Temperament Systems*). Another explanation has to do with tonal theory. In some cases it is preferable to write [D#, C##, D#] instead of [D#, D♮, D#], even on a keyboard, because C## is the leading tone of D#. Another way of saying this is that C## is part of the scale of D# major, while D♮ is not.

Composers like Chopin and Tchaikovsky used double sharps and flats, and any professional musician must be able to read them.

3.4 Octaves

There are many ways to represent pitch and octave. One is to pack them as a single integer, like MIDI, in the format $12o+p$, where o is the octave and p is the pitch. In MIDI, every note on a keyboard is represented by a number from 0 to 127, therefore 0 represents C in the first octave, 12 represents C in the second octave, and so on. Similarly, 1, 13, and 25 represent C# on the first, second, and third octaves. The first octave is the one where C has the frequency of 16.352Hz (the lowest note on most pianos is an A in the same octave with a frequency of 27.5Hz). In MIDI, 60 is central C ($(5 * 12) + 0 = 60$) and 62 is central D ($(5 * 12) + 2 = 62$). It's good to know that in some countries the central octave is 3 or 4 instead of 5. In those systems the lowest octaves will have negative numbers such as -1 and -2. Yuck.

The Python function `divmod` provides an easy way to get the octave and pitch number of a midi note:

```
>>> divmod(62, 12)
>>> (5, 2)
```

Another way is to define a note as an object and have separate attributes for pitch value and octave. We'll do this in chapter *Rests and Notes as Python Objects*.

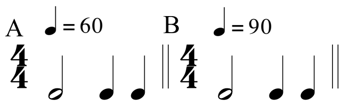
3.5 Note Value

In music notation the relative duration of a note is represented by the note's shape, with the next note in the duration table having half the value of the preceding note. For instance, if a half note lasts 1 second, a quarter note will last 0.5 seconds. The mathematics of note values is simple and beautiful (as usual), and the naming is ugly and confusing (as usual).

Even if I'm not American, I prefer the American way of using half, quarter, eighth, and so on for notes values. I think it's more logical and easier to remember. In other languages, each note value has a name that is not always the same across different languages. For instance, a quarter note is called "black" in French (*noire*) and Spanish (*negra*) but *seminima* in Italian and Portuguese. Don't even start with the British names like "Quasihemidemisemiquaver" (128th). In the following table we can see all note values and durations:

Note	Name	Duration
𝆷	longa	4
𝅜	double whole note (breve)	2
𝅝	whole note	1
𝅗𝅥	half note	1/2
𝅘𝅥	quarter note	1/4
𝅘𝅥𝅮	eighth note	1/8
𝅘𝅥𝅯	sixteenth note	1/16
𝅘𝅥𝅰	thirty-second note	1/32
𝅘𝅥𝅱	sixty-fourth note	1/64
𝅘𝅥𝅲	hundred twenty-eighth note	1/128

We can know the duration in seconds of a note only if we have a tempo in beats per minutes (BPM). For instance, in the following image, the tempo marking of ♩ = 60 tells us that it should have 60 quarter notes per minute, or consequently, that each quarter note should last 1 second. Similarly, the marking of ♩ = 90 indicates that it should have 90 quarter notes per minute, or that each quarter note should last 0.66666 seconds. In this example, if a quarter note lasts 0.6666, a half note should be the double of that time (1.3333s) and a eighth note half of it (0.333333).



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 ♩ = 90), and *t* is the tempo itself.

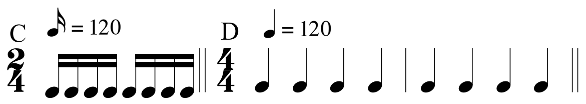
The function `note_duration` returns the time in seconds a note value has in a specific tempo:

```
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 (I'm using `from __future__ import division`):

```
>>> note_duration(1/4, 1/4, 90)
>>> 0.666666666667
>>> note_duration(1/2, 1/4, 90)
>>> 1.33333333333
>>> note_duration(1/8, 1/4, 90)
>>> 0.333333333333
```

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

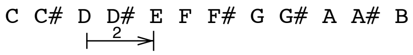
Mathematically, a musical interval is the difference in semitones between two notes. Intervals can also be represented with integers:

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

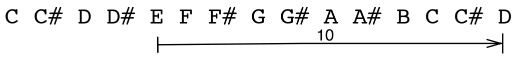
We can see that the interval from D to E is 2 semitones and from E to D is 10 semitones:

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

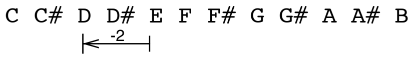
The reason that the same notes in different order will have different intervals has to do with direction. There are 2 semitones from D to the next E:



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



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



And, of course, -2 mod 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]
```

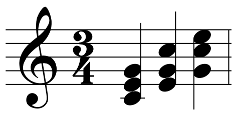
We can see rotation in action in medieval modes where each mode is a rotation of the other:

```
>>> 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

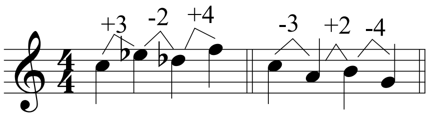
Inversion is used a lot in music, and, as many things in music, it has different meanings.

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

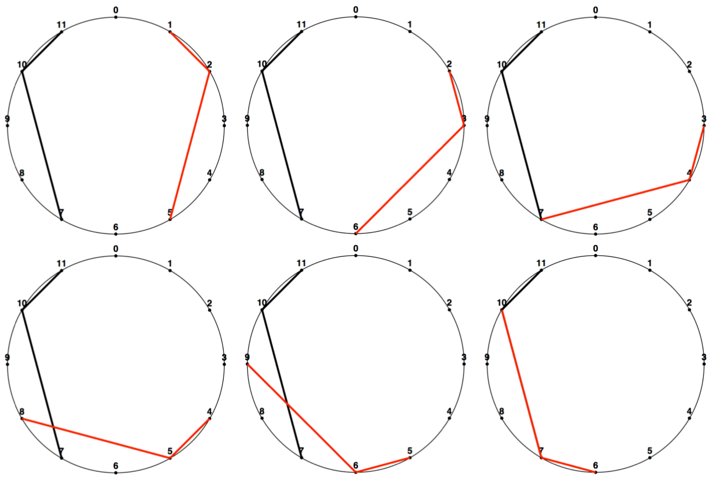


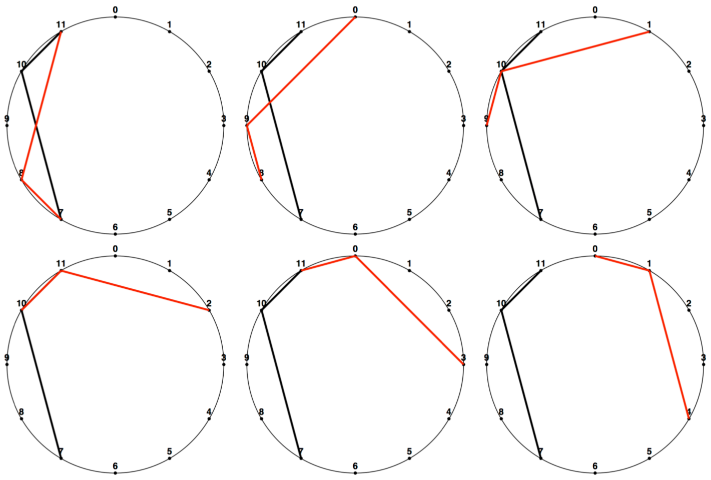
```
>>> rotate([0,4,7], 0)
>>> [0, 4, 7]
>>> rotate([0,4,7], 1)
>>> [4, 7, 0]
>>> rotate([0,4,7], 2)
>>> [7, 0, 4]
```

The inversion of a melody is different. The direction of the original intervals is changed in the opposite direction. For example, if a melody has the intervals +3, -2, +4 (see picture below), its inversion will have the intervals -3, +2, -4. Hence, the inversion of [C, Eb, Db, F] is [C, A, B, G]:



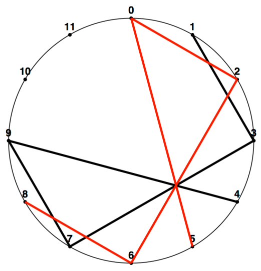
This mirroring operation is known as *reflection* in mathematics. It's easy to grasp the main concept if you think of the set of notes inverted through an axis. In the following images we can see every inversion for the set of notes *a* = [11, 10, 7] (the inversions are in red or as a dotted line, depending on your edition). The first image has the inversion of *a* when the inversion index is equal to 0, the second image has the inversion of *a* when the inversion index is equal to 1, and so on until the last image, when the inversion index is equal to 11:





It's important to notice that the inversion index doesn't correspond to notes or intervals in the integer notation. Because the inversion axis is 0 in the first image, you may think the note 0 is the inversion axis. But in the second image, the axis is 0.5.

By the way, I generated the images above using the function *plot.plot2* in *pyknon*. It's a handy function to inspect inversions visually. Also, the list of notes doesn't have to be sequential; you can use any order you want. In the following image I'm plotting the set of notes [1, 3, 7, 9, 4] and its inversion:



The following function computes the inversion of a list of notes through an inversion index:

```
def inversion(notes, index=0):
    return [mod12(index - n) for n in notes]
```

We can, indeed, verify that the inversion of [11, 10, 7] is [1, 2, 5] when the inversion index is 0:

```
>>> inversion([11, 10, 7], 0)
>>> [1, 2, 5]
```

As with any reflection, music inversion is an *involution*; its inverse function is itself. It's actually an easy concept, but if you want to see some confused music students, go to a classroom and say "the inverse function of an inversion is the inversion." Of course, it's easy to see this in code:

```
>>> chord = [0, 4, 7]
>>> [0, 4, 7]
>>> chord == inversion(inversion(chord))
>>> True
```

To be honest, musicians don't think of inversion through an axis very often; it's much more common and simpler to think of an inversion starting with a note (as in "the inversion of [D, F#, G] starting with A"). The following utility function combines transposition and inversion to implement that:

```
def inversion_startswith(notes, start):
    transp = transposition_startswith(notes, 0)
    return transposition_startswith(inversion(transp), start)

>>> inversion_startswith([0, 3, 1, 5], 3)
>>> [3, 0, 2, 10]
```

3.6.6 Interval Names

Because music since the 1600s evolved from scales, tonalities, and temperament systems, the naming of intervals is a little bit weird. We'll see in chapter *A Look Inside the Primitives* that the same interval such as a minor third can have different sizes (in Hertz) according to the tuning system.

An interval name has a quantity such as "second", or "third", and a quality such as "perfect," "major," or "minor." The following table summarizes the qualities an interval can have:

Quantity	Quality
Unison, Fourth, Fifth	Diminished, Perfect, Augmented
Second, Third, Sixth, Seventh	Diminished, Major, Minor, Augmented

So how come some intervals are “perfect” while others are “minor” and “major”? There are many ways to explain this, none of them fully satisfactory. The ancient Greeks used a system with four notes (a tetrachord) where the outer notes (forming an interval of a perfect fourth) were immutable (and therefore “perfect”) and the inner notes were mutable. Another way of explaining this is that both minor and major intervals are considered consonant, but if you raise or lower an unison, a fourth, or a fifth you'll get intervals that were considered dissonant.

You can see in the following table a list of the main intervals. Also, Wikipedia's entry on interval is good.

semitones	notes	name
0	C–C	perfect unison
1	C–Db	minor second
2	C–D	major second
3	C–Eb	minor third
4	C–E	major third
5	C–F	perfect fourth
6	C–F#	augmented fourth
6	C–Gb	diminished fifth
7	C–G	perfect fifth
8	C–Ab	minor sixth
9	C–A	major sixth
10	C–Bb	minor seventh
11	C–B	major seventh

Let's write a Python function `interval_name` that returns the interval name between two notes. This is how it works:

```
>>> interval_name("C", "Db")
>>> Minor Second
>>> interval_name("Eb", "C#")
>>> Augmented Sixth
```

We define `name_to_diatonic` to get the “diatonic” integer notation. It's the number representation of a note, disregarding accidentals in a diatonic (C major) scale:

```
def name_to_diatonic(note_string):
    notes = "C D E F G A B".split()
    name = note_string[0:1].upper()
    return notes.index(name)

>>> name_to_diatonic("C")
>>> 0
>>> name_to_diatonic("C#")
>>> 0
>>> name_to_diatonic("Db")
>>> 1
>>> name_to_diatonic("D")
>>> 1
```

We use the function `name_to_number` that we defined earlier (in section *Integer Notation*) to get the integer representation of the notes. Let's see the implementation for `interval_name`:

```
def interval_name(note1, note2):
    quantities = ["Unison", "Second", "Third", "Fourth", "Fifth", "Sixth", "Seventh"]
    n1, n2 = name_to_number(note1), name_to_number(note2)
    d1, d2 = name_to_diatonic(note1), name_to_diatonic(note2)
    chromatic_interval = interval(n2, n1)
    diatonic_interval = (d2 - d1) % 7
    quantity_name = quantities[diatonic_interval]
    quality_name = get_quality(diatonic_interval, chromatic_interval)
    return "%s %s" % (quality_name, quantity_name)
```

With `name_to_number` and `name_to_diatonic` we can have a chromatic and a diatonic interval. The diatonic interval is used to get the interval's *quantity* by a simple list lookup. Because the result of `name_to_diatonic` is the position of a note name (without accidentals) in a diatonic scale, the diatonic interval will correspond to a generic interval such as "third" or "fourth," but without any quality (such as "major" or "perfect"). Finally, we get the interval's *quality* with `get_quality`:

```
def get_quality(diatonic_interval, chromatic_interval):
    if diatonic_interval in [0, 3, 4]:
        quality_map = ["Diminished", "Perfect", "Augmented"]
    else:
        quality_map = ['Diminished', 'Minor', 'Major', 'Augmented']

    index_map = [-1, 0, 2, 4, 6, 7, 9]
    try:
        return quality_map[chromatic_interval - index_map[diatonic_interval]]
    except IndexError:
        raise SimpleMusicError("Sorry, I can't deal with this interval :-(")
```

As we have seen, there are two classes of interval quality: the ones that can be perfect, and the ones that can be minor or major. The chromatic interval is enough to determine the interval quality, so `quality_map[chromatic_interval]` should be sufficient. But since `quality_map` has only three or four elements, and the value for `chromatic_interval` can go up to 11, we need to scale its value according to `index_map`.

Let's see how this works with a couple examples. If the notes are C and Db, the value for `chromatic_interval` is 1 (it has one semitone) and the value for `diatonic_interval` is 1 (it's the second item in the diatonic scale). Therefore, the result of `index_map[1]` is 0, and we end up with `quality_map[1 - 0]` that evaluates to "minor."

If the notes are C and Fb, the value for `chromatic_interval` is 4 (it has four semitones) and the value for `diatonic_interval` is 3 (it's the fourth item in the diatonic scale). The result for `index_map[3]` is 4 and `quality_map[4 - 4]` is "diminished," which is expected since the interval between C and Fb is a diminished fourth.

The function `interval_name` will work with all basic intervals (minor, major, perfect, augmented, and diminished) with the exception of the augmented seventh. The reason is that the augmented seventh has the same size in semitones as the octave, but since we're using mod12, the value for `chromatic_interval` is 0 instead of 12. We could fix it with the following code:

```
chromatic_interval = 12 if chromatic_interval == 0 else chromatic_interval
```

but this would break when the interval is a unison. We could fix it with something like:

```
chromatic_interval = 12 if chromatic_interval == 0 else chromatic_interval
index = (chromatic_interval - index_map[diatonic_interval]) % 12
return quality_map[index]
```

But this is ugly as hell, and the fact that we are dealing with two exceptions tell us that we need a better mathematical abstraction. In production I wouldn't use the mod12 integer notation for tonal intervals. I'd use a system that can differentiate flats and sharps such as *(Hewlett, 1992)* or *(Brinkman, 1982)*. On the other hand, `interval_name` and `get_quality` are good for teaching and demonstrating because they mimic how a musician may think about intervals.

Exercise 4. Extend the function `get_quality` to deal with doubly augmented and doubly diminished intervals such as Eb–F## and E–Cbb.

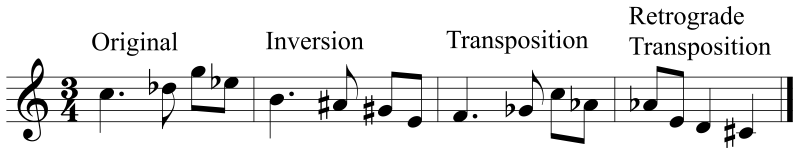
Exercise 5. Read *(Hewlett, 1992)* and implement a function to return the interval name using his numerical system.

3.6.7 Simple combinations

In the chapter *Means of Combination* we'll explore how to make interesting examples (for same value of interesting) using the combination of these primitives and operations. For now, let's have a taste by combining a motif with its inversion, transposition, and retrograde:

```
>>> motif = [0, 1, 7, 3]
>>> [0, 1, 7, 3]
>>> a = inversion_startswith(motif, 11)
>>> [11, 10, 4, 8]
>>> b = transposition_startswith(motif, 5)
>>> [5, 6, 0, 8]
>>> c = retrograde(transposition(motif, 1))
>>> [4, 8, 2, 1]
>>> motif + a + b + c
>>> [0, 1, 7, 3, 11, 10, 4, 8, 5, 6, 0, 8, 4, 8, 2, 1]
```

We concatenate the result in one big list of notes. It doesn't sound very impressive, but if we change octaves and duration, the same list of notes will sound much better:



Track 1. Simple combination. You may notice that the order of the notes number 13 and 14 on the sheet music are swapped in respect to the original list. That is, the first two notes in the last bar should've been E and Ab instead of Ab and E. This is to show that we don't need to follow the same order rigorously, and by swapping the two notes we get to repeat the Ab from the previous bar, which helps to make the phase smoother. See also *Exercise 17* for another example of "wrong notes".

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_interval(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)`?:

```
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 appalling 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 thousands 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.

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_list, 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_list)
        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: [<E>, <E>, <B>, <F>, <E>]>
```

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>, <A>, <A>, <D>, <C>]>
```

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 `ran-`

dom1 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
```

The utility function `pascals_triangle` in `random_combinations.py` will generate each row as a list. For instance, to generate the first five rows:

```
>>> list(pascals_triangle(5))
>>> [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]
```

There's something funny about transforming the Fibonacci sequence into music. Here are the first 25 numbers in the sequence: 0, 1, 1, 2, 3, 5, 8 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657. If we apply `mod12` on them we'll get the following notes: 0, 1, 1, 2, 3, 5, 8, 1, 9, 10, 7, 5, 0, 5, 5, 10, 3, 1, 4, 5, 9, 2 11, 1. If we apply `mod12` on the next 25 numbers in the Fibonacci sequence we will get the same list of notes: 0, 1, 1, 2, 3, 5 8, 1, 9, 10, 7, 5, 0, 5, 5, 10, 3, 1, 4, 5, 9, 2, 11, 1. That is, the Fibonacci sequence mod 12 is cyclical! Try to listen to this repetition in the sound example below.

In the following function we make one list with the first 55 Fibonacci numbers and another with the first 30 rows in Pascal's triangle. We use these numbers as input for the `play_list` function:

```
def random_fib():
    octave = range(5, 7)
    fib = fibonacci(100000000000)
    pascal = flatten(pascals_triangle(30))

    n1 = play_list(fib, octave, 1/16)
    n2 = play_list(pascal, 4, 1/16)
    n3 = play_list(pascal, octave, 1/16)

    gen_midi(”fibonnacci.mid”, n1)
    gen_midi(”pascal.mid”, n2)
    gen_midi(”pascal_octaves.mid”, n3)
```

Track 5. The Fibonacci sequence.

Track 6. Pascal's Triangle.

Track 7. Pascal's Triangle in octaves.

To be honest, when I was writing the code I thought Pascal's triangle was going to sound boring due to the repetitions. But it turns out that I like it very much! (Don't hold that against me). One more proof that repetition is good.

Exercise 10. Input your favorite integer sequence in `play_list` and see how it sounds. If you don't have a favorite integer sequence, go to The On-Line Encyclopedia of Integer Sequences (https://oeis.org/) and pick one. What kind of person doesn't have a favorite integer sequence?

5.3 Combination of Notes Horizontally

Now that we have heard how some randomly generated music sounds, let's combine some primitives to make something more interesting. To me, one of the best things about applying programming to music is that we can describe and reproduce the *musical process* involved.

We'll reproduce the basic process of four compositions: *Piano Phase* (1967) by Steve Reich, *Crab Canon* and *Quaerendo invenietis* (1747) from the *Musical Offering* by J. S. Bach, and *Agnus Dei* (1489?) by Josquin des Prez. It's interesting that these pieces are more than 200 years apart, the older being more than 500 years old, and yet they still manage to use the same primitives we have seen so far.

5.3.1 Steve Reich, *Piano Phase*

The musical process in *Piano Phase* is very simple. It's a composition for two pianos where one pianist plays a 12-note pattern repeatedly and unchangeably, while the other pianist plays rotations of the original pattern. Each rotation is repeated a number of times.

Track 8. *Piano Phase* played by humans. (Only in your resources webpage, sorry.)

This process can be implemented in code in many ways. The following implementation is not the simplest, but the number of rotations and the number of repetitions of each rotation is abstracted in the function piano_phase; we can have as many rotations and repetitions as we want. The function gen_patterns makes a NoteSeq with the appropriate rotations and repetitions of pattern.

```
def gen_patterns(pattern, number_rotations=12, repeat=4):
    result = NoteSeq()
    for n in range(0, number_rotations):
        rotation = pattern.rotate(n)
        result.extend(rotation * repeat)
    return result

def piano_phase(number_rotations=12, repeat=4):
    pattern = NoteSeq("E16 F# B C#'' D'' F#' E C#'' B' F# D'' C#")
    piano1 = pattern * (number_rotations + 1) * repeat
    piano2 = gen_patterns(pattern, number_rotations, repeat)

    midi = genmidi.Midi(2, tempo=108)
    midi.seq_notes(piano1)
    midi.seq_notes(piano2, track=1, time=3*repeat)
    midi.write("midi/piano-phase.mid")
```

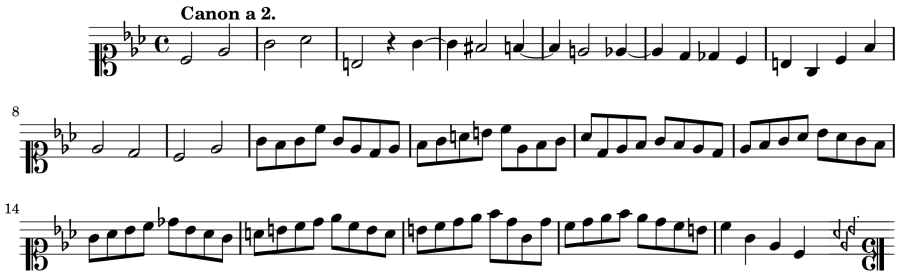
Track 9. *Piano Phase* generated by our function. Naturally, this one sounds much more mechanical and precise.

I find it remarkable that we can implement and describe the musical process of a composition in a dozen lines of code. It's even more remarkable that the gist of the process is just two lines of code:

```
piano1 = pattern * (number_rotations + 1) * repeat
piano2 = gen_patterns(pattern, number_rotations, repeat)
```

5.3.2 J. S. Bach, *Crab Canon*

The *Crab Canon* from Bach's *Musical Offering* is a composition for two instruments, but Bach wrote it in only one line. One musician plays the first voice normally while the other plays the score backwards. Yes, backwards. I do that when I drink too much and have a violin in my hands. You don't want to be near me. There's a nice Youtube video that shows this canon on a Möbius strip: J.S. Bach - Crab Canon on a Möbius Strip.



The implementation is very straightforward. One voice plays the theme unchanged while the other plays the theme's retrograde one octave below. It's that simple:

```
def crab_canon():
    theme2 = NoteSeq("file://canon-crab")
    rev_theme = theme2.transposition(-12).retrograde()

    midi = Midi(2, tempo=120)
    midi.seq_notes(theme2)
    midi.seq_notes(rev_theme, track=1)
    midi.write("midi/canon-crab.mid")
```

Although it's not in the score, it's common to play the second voice one octave below the original theme to make it easier to hear both voices.

Track 10. Crab canon generated by our function. (Check our resources webpage to listen to a human performance)

One could argue that we are cheating since we are not *generating* the theme (the full theme is encoded in the file canon-crab). While this is true, the compositional process is fully described in the code above. If we change the theme and run the code we'll get a different composition with the same process. In fact, you could substitute canon-crab with your own file.

Exercise 11. Create your own crab canon using the function crab_canon.

5.3.3 Bach, Canon *Quaerendo invenietis*

Now let's see the code to implement another canon in the *Musical Offering*. I won't show the sheet music here, but from implementation you can see that it's a canon by inversion:

```
def canon():
    theme1 = NoteSeq("file://canon-quaerendo-invenietis")
    part1 = theme1 + theme1[2:] + theme1[2:11]
    part2 = theme1 + theme1[2:] + theme1[2:4]

    voice1 = part1
    voice2 = part2.inversion_startswith(Note(2, 4))

    midi = Midi(2, tempo=150)
    midi.seq_notes(voice1, time=3, track=0)
    midi.seq_notes(voice2, time=13, track=1)
    midi.write("midi/canon.mid")
```

Since the theme is repeated, we need to do some fiddling to make the repetitions have the appropriate size (when we define `part1` and `part2`). However, the gist of the function is just two lines; the theme and its inversion:

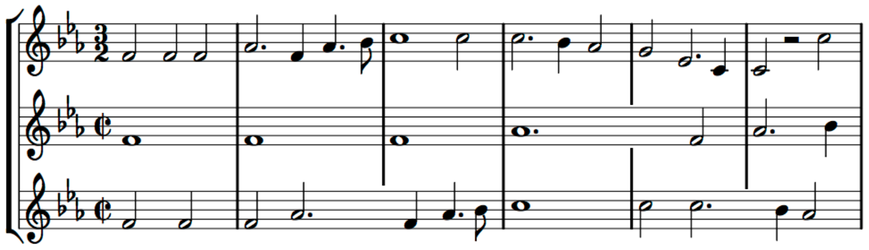
```
voice1 = part1
voice2 = part2.inversion_startswith(Note(2, 4))
```

Track 11. Canon *Quaerendo invenietis*

Exercise 12. Change the `canon` function to have a different process. Instead of inversion, use transposition, retrograde, or other operations and see how it sounds.

5.3.4 Josquin des Prez, *Agnus Dei*

The *Agnus Dei* by Josquin des Prez is a prolation canon, where a melody is accompanied by one or more imitations of itself (possibly transposed or inverted) in other voices at different speeds. Here are the first six measures of Josquin's *Agnus Dei*:



Track 12. Josquin's *Agnus Dei*

Do you notice a pattern? What if we write the note values:

```
S: 1/2 1/2 1/2 3/4 1/4 3/8 1/8
A: 1   1   1   3/2 1/2 3/2 1/4
T: 1/2 1/2 1/2 3/4 1/4 3/8 1/8
```

As you can see all three voices have the same rhythmic pattern, but the values for the second voice are the double of the others. The first and third voices seem to have the same notes values, but in the sheet music one appears to be moving faster than the other. Josquin uses a weird (to us) notation to indicate that the first voice should have three notes at the same time the second voice has two notes of the same value. He uses different time signatures; today we'd write the same thing using tuplets.

If this sounds confusing, just look at the following code to implement Josquin's *Agnus Dei* (the method `transp` is an alias to `transposition_startswith`):

```
def josquin():
    main_theme = NoteSeq("file://josquin")
    theme1 = main_theme.stretch_dur(0.66666)
    theme2 = main_theme[0:24].stretch_dur(2).transp(Note("C"))
    theme3 = main_theme[0:50]

    midi = Midi(3, tempo=80)
    midi.seq_notes(theme1, track=0)
    midi.seq_notes(theme2, track=1)
    midi.seq_notes(theme3, track=2)
    midi.write("midi/josquin.mid")
```

We have three tracks with the theme. In the first and second tracks the theme is stretched by 2/3 and 2, respectively. The duration of the third track is not stretched. The second track also transposes the theme to start with the note C. It's simple, isn't it?

In these four examples we have seen that it's possible to generate meaningful music with a small amount of code. The key is to capture the *compositional process* of the piece. Naturally, many compositions will have processes much more elaborate than the ones we have used; too complex to fit in a few lines of code. But the operations we have seen in chapter *The Primitives of Music* are the foundation for many combinations in music and have been used for hundreds of years.

5.4 Chords: Combination of Notes Vertically

How many chords exist? Let's start with three-note chords. As you remember, a combination is a way of selecting *k* things from a larger group *n* where order doesn't matter. For example, in how many ways can we select two items from (a, b, c)? The answer is three:

```
(a, b)
(a, c)
(b, c)
```

Items like (b, a) and (c, b) don't count, since order doesn't matter (they'd matter if we were counting *permutations*). The formula for combinations is as follow:

$$C(n,k) = \frac{n!}{(n-k)!k!}$$

So, how many three-note chords can we have? If we follow the formula for selecting groups of 3 notes from 12 we'll have 220 chords. It's easy to implement a small utility function to do the calculation for us (`factorial` is defined in the `math` module):

```
def chord_combinations(n, k):
    return factorial(n) / (factorial(n - k) * factorial(k))

>>> chord_combinations(12, 3)
>>> 220
```

If you are feeling adventurous (or bored), you can generate every one of the 220 combinations with `itertools.combinations`:

```
>>> list(combinations(range(0, 12), 3))
```

This should answer our question, but another question is "how many *different* types of chords exist?" If we take a close look at these 220 combinations we'll see that we have chords related by transposition such as (0, 4, 7) and (1, 5, 8), that is, C major and Db major, respectively. We also have chords related by inversion such as (0, 4, 7) and (0, 8, 5). We want to remove those duplications.

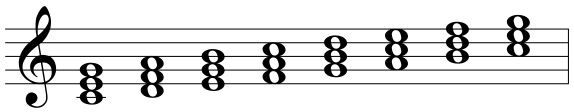
We could write code to do that for us, but a simpler way is to use *pitch class sets*. Allen Forte classified every subset from a twelve-tone collection, removing the subsets related by transposition or inversion *(Forte, 1973)*. In the end, we actually have only twelve different subsets for three elements. The module `pyknon.pcsets` has all possible pitch class sets in the dictionary `PC_SETS`. The key in the dictionary is the set's Forte number and the value is the set itself. These are the three-note subsets:

```
'3-1': [0, 1, 2],
'3-2': [0, 1, 3],
'3-3': [0, 1, 4],
'3-4': [0, 1, 5],
'3-5': [0, 1, 6],
'3-6': [0, 2, 4],
'3-7': [0, 2, 5],
'3-8': [0, 2, 6],
'3-9': [0, 2, 7],
'3-10': [0, 3, 6],
'3-11': [0, 3, 7],
'3-12': [0, 4, 8],
```

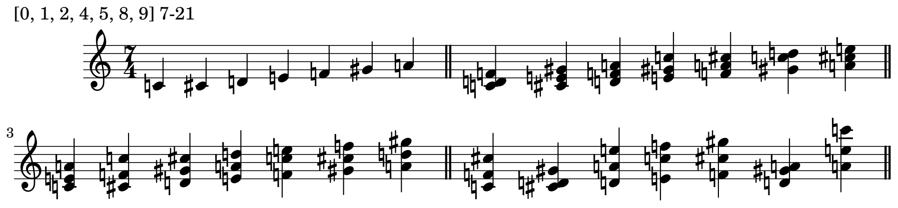
Some people don't like to remove the sets related by inversion, since they sound different (for instance, the inversion of a major chord is a minor chord). Larry Solomon's Table of Pitch Class Sets. has nineteen three-note sets instead of twelve. In any case, the main point is that because of some music operations such as transposition, we went from 220 possible chords to only 12 (or 19). This is something that happens quite often when dealing with musical data; a larger mathematical data is reduced due to music operations.

Exercise 13. Write a Python function that filters all three-note chords from the 220 combinations and returns only the ones that are not related by transposition.

Now that we have seen how many chords we have, let's see how to harmonize every scale. The harmonization for the major scale is pretty simple; you stack notes vertically following every other note in the scale:



It's fairly common in modern music to use non-traditional scales to generate new and interesting harmonies. For instance, in the example below we have a seven-note scale that is very different from the good ol' major scale. Right after the scale we have harmonizations for every three, four, and five note of the scale. Also notice that we break the thing about having the interval in which we pick notes be the same as the horizontal notes (that is, having chords by thirds, fourths, and so on). This leads to very interesting harmonies.



Track 13. Harmonization

So, to harmonize every scale we need to know how many scales exist, and the answer will vary depending on whom you ask. In a way, a scale is an ordered set of notes, so we can use Allen Forte's pitch class sets.

As I mentioned previously, some people think that Forte's way is too condensed and doesn't take into account a bunch of scales, but it'll do for the purpose of this section (There are 4,095 possible combinations of scales, but there are only 208 pitch class sets).

First we implement the method `harmonize` in the class `Note`. It accepts the following arguments:

- `scale`: an iterable containing `Notes`.
- `interval`: the interval quantity between notes in the chord. For instance, a regular triad has two thirds pilled up, therefore the value for `interval` will be 3.
- `size`: the number of notes in a chord.

The method `harmonize` will compute the indices in the scale that correspond to the notes to be harmonized. For example, if we want to harmonize the note D in the C major scale the indices will be 1, 3, and 5. Finally, `harmonize` calls `tonal_transposition` on each of those indices to get the harmonized chord:

```
def harmonize(self, scale, interval=3, size=3):
    i = (interval - 1)
    indices = range(1, size*i, i)
    return [self.tonal_transposition(x, scale) for x in indices]
```

While a regular transposition (as defined in section *Transposition*) is the sum of a note and a transposition index, a *tonal* transposition looks like an index in a table, where the table is the scale in question. For example, the (regular) transposition of C with the transposition index 3 (a minor third above) is Eb, while the tonal transposition of C a third above in the scale of A minor is E and in the scale of C minor is Eb. The method `tonal_transposition` is slightly complicated because we need to transpose the octave as well:

```
def tonal_transposition(self, index, scale):
    pos = index + scale.index(self) - 1
    octave, rest = divmod(pos, 7)
    note = copy.copy(scale[pos % len(scale)])
    note.octave += octave
    return note
```

Here's how we use `harmonize`:

```
>>> c = Note("C")
>>> <C>
>>> c_major_scale = NoteSeq("C D E F G A B")
>>> <Seq: [<C>, <D>, <E>, <F>, <G>, <A>, <B>]>
>>> c.harmonize(c_major_scale)
>>> [<C>, <E>, <G>]
```

The implementation of the method `harmonize` in the class `NoteSeq` is straightforward:

```
    def harmonize(self, interval=3, size=3):
        return [NoteSeq(note.harmonize(self, interval, size)) for note in self]
```

Here's an example of how to harmonize a sequence of notes:

```
>>> c_major_scale = NoteSeq("C D E F G A B")
>>> <Seq: [<C>, <D>, <E>, <F>, <G>, <A>, <B>]>
>>> c_major_scale.harmonize()
>>> [<Seq: [<C>, <E>, <G>]>, <Seq: [<D>, <F>, <A>]>, <Seq: [<E>,
>>> <G>, <B>]>, <Seq: [<F>, <A>, <C>]>, <Seq: [<G>, <B>, <D>]>,
>>> <Seq: [<A>, <C>, <E>]>, <Seq: [<B>, <D>, <F>]>]
```

And the result of `harmonize` will be a list with the harmonization of every chord in the scale. In the next example I'm using 4 as the chord size to generate tetrads:

```
>>> c_major_scale = NoteSeq("C D E F G A B")
>>> <Seq: [<C>, <D>, <E>, <F>, <G>, <A>, <B>]>
>>> c_major_scale.harmonize(size=4)
>>> [<Seq: [<C>, <E>, <G>, <B>]>, <Seq: [<D>, <F>, <A>, <C>]>,
>>> <Seq: [<E>, <G>, <B>, <D>]>, <Seq: [<F>, <A>, <C>, <E>]>, <Seq:
>>> [<G>, <B>, <D>, <F>]>, <Seq: [<A>, <C>, <E>, <G>]>, <Seq: [<B>,
>>> <D>, <F>, <A>]>]
```

Finally, lets generate tetrads separated by fourths instead of thirds:

```
>>> c_major_scale = NoteSeq("C D E F G A B")
>>> <Seq: [<C>, <D>, <E>, <F>, <G>, <A>, <B>]>
>>> c_major_scale.harmonize(interval=4, size=4)
>>> [<Seq: [<C>, <F>, <B>, <E>]>, <Seq: [<D>, <G>, <C>, <F>]>,
>>> <Seq: [<E>, <A>, <D>, <G>]>, <Seq: [<F>, <B>, <E>, <A>]>, <Seq:
>>> [<G>, <C>, <F>, <B>]>, <Seq: [<A>, <D>, <G>, <C>]>, <Seq: [<B>,
>>> <E>, <A>, <D>]>]
```

To harmonize every scale we just create a `NoteSeq` from the numbers in `pc_set` and call the method `harmonize`:

```
for forte, pc_set in pcset.PC_SETS.items():
    scale = NoteSeq([Note(n) for n in pc_set])
    scale.harmonize()
```

Generating harmonizations for every scale is interesting, but it starts to get interesting when we can filter things. For instance, we may want to generate harmonizations for the scales that have more than four notes and have consecutive intervals greater than one semitone. This can be accomplished with the function `filter_sets`. This is how we'd express the previous example:

```
>>> filter_sets(lambda intervals, size: size > 4 and 1 not in intervals)
```

The function `filter_sets` accepts an anonymous function with a list of the consecutive intervals in a set and the size of a set as parameters. If the condition in the body of the anonymous function is met the set is returned:

```
def filter_sets(condition, all_sets=pcset.PC_SETS):
    sets = {}
    for forte, pc_set in all_sets.items():
        intervals = simplemusic.intervals(pc_set)
        size = len(pc_set)
        if condition(intervals, size):
            sets[forte] = pc_set
    return sets
```

This is an example of how high-order functions can be used to abstract code. Also, the built-in function `all` is very handy to chain conditions. Let's say we want to get scales that have more than four notes and have only one consecutive semitone and one consecutive whole tone:

```
>>> filter_sets(lambda i, s: all([s > 4, i.count(1)==1, i.count(2)==1]))
>>> {'5-32': [0, 1, 4, 6, 9], '5-31': [0, 1, 3, 6, 9]}
```

As you can see, there's nothing magical about the major scale and the way it's harmonized. It's one among hundreds of scales and dozens of ways of stacking notes on top of each other. And, unsurprisingly, the code to harmonize every scale is relatively simple.

Exercise 14. The method `harmonize` stacks notes by a fixed interval (by thirds, fourths, etc). Create a new version that accepts and uses a "chord template" to harmonize a scale.

5.5 Summary

These are the main points we have seen in the chapter:

- Full randomness doesn't sound good.
- Repetition and limitation of primitives create more familiar sounds.
- We can replicate the musical process of great music in simple code.
- We can generate music using a few operations.
- An operation is a function that maps a set of notes to another set.
- Traditionally we have operations like transposition, inversion, and retrograde, but you can invent your own.
- Operations can be used to create combinations.
- Transposition and inversion reduce the number of combinations.
- We have 4,095 combinations, but only 208 pitch-class sets.

CHAPTER 6

A Look Inside the Primitives

In the previous chapters we assumed that notes and pitches were black boxes. In this chapter we're going to open that black box and see what they are made of.

6.1 The Basics of Sound

As many of you science geeks know, a sound source vibrating will disturb the air, making it oscillate. The air's vibration will in turn make ours eardrums oscillate at the same frequency as the sound source. This sound source can be a guitar string, a car horn, or your bloody noisy neighbor (keep it down Jimmy, will you?). We can't see the air oscillating, but if you put something vibrating in water it's easy to see how the vibration forms waves. There are cool videos on the Internet demoing the physical properties of sound. Search for something like "sound wave experiments" or check this book's resources web page.

If a sound source is vibrating 262 times a second we say that the frequency of its oscillation is 262 cps (cycles per second) or 262 Hz (hertz). We listen to the frequency of a sound as pitch. For instance, we listen to a sound with a frequency of 262 Hz as a middle C. Keep in mind that a sound frequency is an objective property, however the way we perceive pitches is not.

The relationship between pitch and frequency is not linear. For instance, C1 = 32.7 Hz and D1 = 36.7 Hz, a difference of only 4 Hz, while C8 (4,186 Hz) and D8 (4,698 Hz) have a difference of 512 Hz. Actually, the way we perceive musical intervals is logarithmic. For example, the interval between the notes with frequency of 110 Hz and 220 Hz is perceived as the same interval as the notes with 220 Hz and 440 Hz.

So, we listen to a sound with a frequency of 262 Hz as a middle C. How about a sound with a frequency of 263.7 Hz or 264.37 Hz? We listen to all three sounds as a middle C as well but with different intonation or *shades*, if you will. There's a range of frequencies that we hear as the same pitch. The next note, C#, has a frequency of 277.18 Hz. The mapping from frequencies to note names has changed over the years. Today the frequency of a central A is defined by the ISO 16 standard as having 440 Hz. However some orchestras tune the A to 442 Hz and sometimes as high as 444 Hz.

The following functions are useful to calculate the frequency of notes:

```python
from math import log

SEMITONE = 1.059463
NOTES = ['A', 'A#', 'B', 'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#']

def freq_to_note(x):
    interval = int(round(log(x/440.0, SEMITONE))) % 12
    return NOTES[interval]

def note_to_freq(n):
    if n in NOTES:
        return 440 * (SEMITONE ** NOTES.index(n))
```

As we can see, the frequencies of 440, 442, and 449 Hz are all identified as A. The next chromatic note, A#, has a frequency of 466.16 Hz:

```
>>> freq_to_note(440)
>>> A
>>> freq_to_note(442)
>>> A
>>> freq_to_note(449)
>>> A
>>> freq_to_note(455)
>>> A#
>>> note_to_freq(”A#”)
>>> 466.16372
```

But the best part is that we can play with these concepts in practice. I've used Csound to generate the following examples. Csound is a sound design, audio synthesis, and signal processing system freely available at http://csounds.com.

In the next example we are going to hear 11 notes separated by 1 Hz each. The first note has a frequency of 440 Hz, the second 441, and so on until the last note with a frequency of 450 Hz. See how many different notes you can hear. Unless you're a frog (you never know), you are *probably* not going to hear 10 separate notes. When I play this exercise, most people hear from one to three notes, but your mileage may vary.

Track 14. Ten notes separated by 1 Hz.

We humans can hear notes from 20 to 20,000 Hz (again, disregard this if you're a frog), therefore we can't hear low frequencies such as 1 or 2 Hz. But one neat trick is to take advantage of *beats*, the interference between two sounds with close frequencies. In the next examples we are going to hear two simultaneous notes separated by 1, 2, and 3 Hz. Since Hertz is a measure of cycles per second, we should hear the difference between these sounds as 1, 2, and 3 cycles per second:

Track 15. Two simultaneous notes separated by 1 Hz.

Track 16. Two simultaneous notes separated by 2 Hz.

Track 17. Two simultaneous notes separated by 3 Hz.

6.2 The Harmonic Series: a Building Block

We can describe a complex tone such as your voice, a guitar, or a bird as a combination of many simpler periodic waves, that is, sine waves. Each of these sine waves is called a partial with its own frequency, amplitude, envelope, and phase. We usually perceive the pitch of a note as the lowest partial, the fundamental frequency. The relative strength of each partial helps to determine the musical timbre, but other things such as noise are important as well. Here are same basic definitions:

Partial. One of the sine waves that describes a complex tone.

Harmonic. Partials that are related to the fundamental frequency by whole number multiples. This includes the fundamental frequency.

Overtone. Any partial with the exception of the fundamental frequency.

Inharmonic. Partials that aren't related to the fundamental frequency by whole number multiples. Most instruments have some degree of inharmonicity, but instruments such as cymbals and gongs have more.

Some instruments have more harmonic partials than others. Instruments such as cymbals, gongs, and tam-tams are full of inharmonic partials. Some instruments, such as high-pitched flutes and ocarinas have almost no overtones.

The harmonic series is an infinite arithmetic series defined by $\sum_{n=1}^{\infty} fn$, where f is the fundamental frequency and n is the number of partials. For instance, if the fundamental frequency is 100 the first 5 partials will be 100, 200, 300, 400, and 500. A function to calculate the harmonic series of a given fundamental is, of course, straightforward:

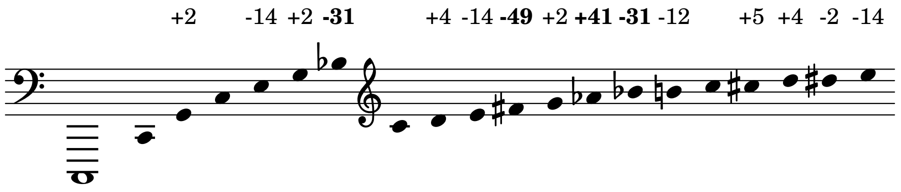
```python
def harmonic_series(n, fundamental):
    return [fundamental * (x + 1) for x in range(n)]
```

Here we calculate the first thirteen partials of the harmonic series that has A as the fundamental:

```
>>> harmonic_series(13, 55)
>>> [55, 110, 165, 220, 275, 330, 385, 440, 495, 550, 605, 660, 715]
```

As we can see, the difference between consecutive partials is constant and equal to the fundamental frequency. However, because we perceive frequencies logarithmically, we hear the higher partials as smaller intervals than the lower ones.

In the following picture we can see an approximation of the harmonic series in musical notation. The numbers on the top indicate the differences from equal temperament in cents, and this difference shows that the equal temperament is not very "natural" after all.



Track 18. The first twenty harmonics in the harmonic series. Notice how the last notes sound different from the equal temperament.

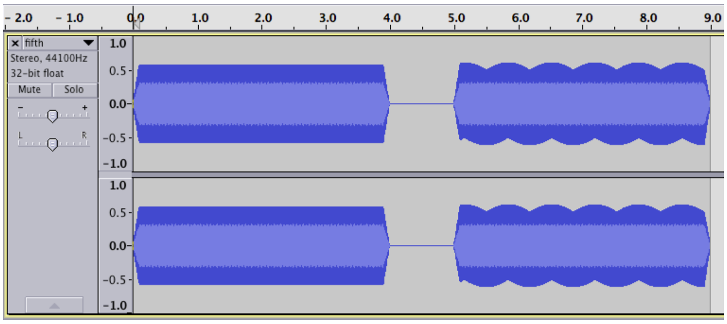
Track 19. As we saw, a complex sound is composed of simpler sounds following the harmonic series. See what happens when we play the previous twenty harmonics at the same time. It sounds like a buzzer, doesn't it? Right now each harmonic is static; it has a fixed amplitude from start to end. To have a more realistic (and pleasant) sound we'd have to vary the volume of each partial, which we call an envelope.

Track 20. In this track the amplitude of each harmonic is reduced as it gets higher. That is, the first note has an amplitude of 80 dB, the second 70 dB, and so on.

Track 21. Some instruments don't have all partials. The clarinet, for instance, has only the odd partials. In this track we simulate that by playing the same harmonic series as before, but filtering the even partials. As you can see, it sounds clarinet-like, although far from realistic.

Track 22. In this track we have a few gong-like sounds from the Amsterdam Catalog of Csound Intruments, based on Risset's "Introductory Catalogue of Computer Synthesized Sounds." The first note has the following partials in Hertz: 240, 277, 340, 385, 605, 670, 812. As we can see, they're not related by whole number multiples, therefore they're inharmonics.

A characteristic of intervals whose frequencies have a whole number ratio is that they don't beat. In the next image we can see a pure fifth (in the harmonic series) and an equal tempered fifth. Notice that the second one has a very pronounced beating:



Now let's listen to some intervals in both the harmonic series and equal temperament. See if you can hear any beating in these intervals:

Track 23. Third.

Track 24. Fifth.

Track 25. Seventh.

Track 26. Octave.

Exercise 15. Try to construct a scale using only frequencies from the harmonic series. Notice how it sounds different from the equal temperament.

6.3 The Beautiful Math of Temperament Systems

The octave is the most basic interval (it's the first interval in the harmonic series), and as we saw, has a ratio of 2:1. The next basic interval in the harmonic series, a fifth, has the ratio of 3:2. We can derive the frequency of every note using this ratio:

```
C     G     D     A     E     B    F#    C#    G#    D#    A#     E#   B#
  3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2 * 3/2
```

This type of temperament is known as the Pythagorean temperament.

The problem of using (only) pure fifths to construct a scale is that the octave can't be divided into 12 fifths. The octave won't "close." If we multiply 3/2 twelve times (there are 12 fifths in one octave), we get 531441/4096 instead of 2/1 (a pure octave). As we can see, the difference between the lowest C and the upper B# is 1.746 Hz. This difference is known as the Pythagorean comma:

```
>>> o = 2**7
>>> 128
>>> n = Fraction(3, 2)**12
>>> 531441/4096
>>> float(n)
>>> 129.746337891
>>> float(n - o)
>>> 1.74633789062
```

This is one of the main reasons that composers, theorists, and performers have created so many tuning systems over the years. Some tuning systems relinquish some pure fifths to have more pure thirds, while others renounce everything (equal temperament). Here are the ratios for the Pythagorean temperament:

note	ratio
C	1/1
C#	2187/2048
D	9/8
D#	32/27
E	81/64
F	4/3
F#	729/512
G	3/2
G#	6561/4096
A	27/16
A#	16/9
B	243/128

The next interval in the harmonic series is the just third with the ratio of 5:4. Here's a simple C major scale with both the just temperament and Pythagorean. Notice the major third from C to E is bigger in the just temperament (5/4 > 81/84):

	Just	Pythagorean
C	1/1	1/1
D	9/8	9/8
E	5/4	81/84
F	4/3	4/3
G	3/2	3/2
A	5/3	27/16
B	15/8	243/128
C	2/1	2/1

Now the question that torments most music students I have ever met: "How come D# is different from Eb?" In the 1600s and 1700s they would tune a harpsichord in the key of the composition to be played. So, in a recital, if a piece was in D major, they'd tune it in a way that it'd sound the most consonant for D major. If the next piece was in F major, tough luck, they'd retune the whole keyboard.

It's useful to have a Python function to generate a scale given a temperament and a base frequency:

```
def scale_freqs(name, base_freq=440):
    return [float(x * base_freq) for x in name]
```

The argument `name` is a list of fractions that define a temperament. For example, the definition for the just intonation temperament is as follows (I import `fractions.Fraction` as `F`):

```
just_intonation = [F(1,1), F(16,15), F(9,8), F(6,5), F(5,4), F(4,3),
                   F(7,5), F(3,2), F(8,5), F(5,3), F(9,5), F(15,8)]
```

If we tune our keyboard using the just temperament in C, we will have the following frequencies:

```
>>> c = scale_freqs(just_intonation, 528)
>>> [528.0, 563.2, 594.0, 633.6, 660.0, 704.0, 739.2, 792.0, 844.8,
>>> 880.0, 950.4, 990.0]
```

In the following code we have two chromatic scales, C and E in the just temperament. We can see that Eb in the scale of C has a frequency of 633.6 Hz while the enharmonically equivalent D# has a frequency of 618.75 Hz if tuned in E (we divide the frequency by two to adjust the octave):

```
>>> c = scale_freqs(just_intonation, 528)
>>> [528.0, 563.2, 594.0, 633.6, 660.0, 704.0, 739.2, 792.0, 844.8,
>>> 880.0, 950.4, 990.0]
>>> e = scale_freqs(just_intonation, 660)
>>> [660.0, 704.0, 742.5, 792.0, 825.0, 880.0, 924.0, 990.0,
>>> 1056.0, 1100.0, 1188.0, 1237.5]
>>> c[3]
>>> 633.6
>>> e[11] / 2
>>> 618.75
```

This happens because in most temperaments each interval has a different size. Equal temperament is, of course, the only temperament where every interval has the same size.

Here's another way of thinking. In the following table we have a simple C major scale in the just temperament. I want to know what frequency E will have in the C minor scale. One way to calculate it is to multiply the ratio for the minor third. We have four minor thirds in this scale: between B and D, D and F, E and G, and A and C. Their ratios are 5/3, 32/27, 6/5, and 6/5, respectively.

name	ratio	freq
C	1/1	528.0
D	9/8	594.0
E	5/4	660.0
F	4/3	704.0
G	3/2	792.0
A	5/3	880.0
B	15/8	990.0
C	2/1	1056

If we decide that the ratio between C and E should be 6/5, then E will have a frequency of 633.6 Hz (528 * 6/5). Now let's have a major third from B in the same scale. If we use the interval 5/4 for the major third, then D# will have a frequency of 618.75 Hz (990 * 5/4 and divided by 2 to fit in the octave). As we can see, Eb and D# don't have the same frequency in this example.

Here are some examples of scales and intervals played in different tuning systems:

Track 27. Kirnberger tuning.

Track 28. Pythagorean tuning.

Track 29. Just intonation.

Track 30. Equal temperament.

In this chapter we've seen the basics of sound, especially how pitches relate to frequency, how complex sounds are composed of simpler sounds, and how on earth D# is different from Eb. But we barely scratched the surface of these subjects. To know more check out our resources webpage.

CHAPTER 7

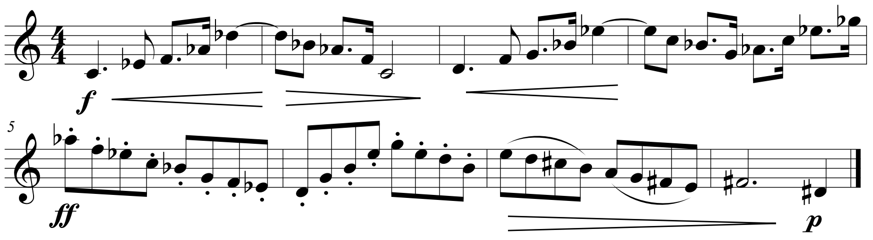
Means of Abstraction

In this section we'll look at how to combine small elements into longer sections.

In programming, we have many ways to combine and abstract small elements into bigger pieces. We may combine expressions into functions and methods, methods into classes, classes and functions into modules, and modules into packages. One important aspect is *naming* things. When we have the same piece of code repeated, we can abstract it in a function or in a class. And by giving it a name we can use it again. To quote Gerald J. Sussman, "If you have the name of the spirit you have power over it" (http://bit.ly/sussman2).

If this sounds like Computer Science 101, it's because it is. But composers have used a similar scheme to organize their compositions. Basic elements of music can be combined in motifs, which can be grouped in phrases, which can be used to form periods, which will create sections, which will form movements, which will be part of a whole 40 minute symphony.

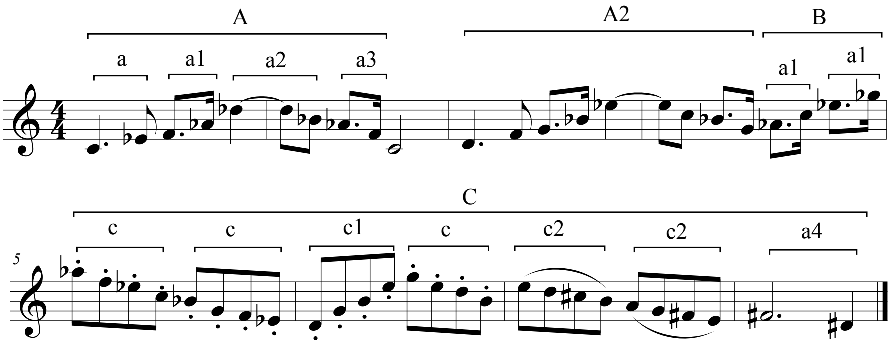
Let's take a look at the following music excerpt:



We can notice some repetition going on but there are also things that sound different (for instance, measures one and five).

Track 31. Simple music excerpt.

If we mark the things that are similar with letters, we may end up with something like the following image. Letters like *a1* and *a2* indicate that both elements are the same motif but with different variations or operations.



Motifs like *a* and *a1* have the same interval (a minor third) and the same rhythmic structure. In fact, *a1* is just like *a* but transposed and stretched. We saw these operations in chapter *Means of Combination*.

We can see the code used to generate this excerpt below. In this example I'm using `transp` as an alias for `transposition_startswith` and `inv` as an alias for `inversion_startswith`. Notice how every *an* is *a* transposed, inverted, or stretched. *c* looks like a different motif, but in fact *c* is the combination of *a* and *a1* with the rhythm changed (line 11).

```
 1  def abstraction():
 2      a = NoteSeq(”C4. Eb8”)
 3      a1 = a.transp(5).stretch_dur(0.5)
 4      a2 = a.inv(”Db’’”)
 5      a3 = a1.inv(8)
 6
 7      A = a + a1 + a2 + a3
 8      A2 = A.transp(2)
 9      B = a1.transp(8) + a1.transp(”Eb’’”)
10
11      c = NoteSeq([Note(x.value, dur=0.125) for x in a + a1])
12      C = (c.inv(”Ab’’”) +
13           c.inv(10) +
14           c.stretch_interval(2).transp(2) +
15           c.inv(”G’’”) +
16           c.inv(”E’’”).stretch_interval(1) +
17           c.inv(”A”).stretch_interval(1)
18           )
19
20      a4 = a.stretch_dur(2).inv(6)
21
22      Part1 = A + NoteSeq(”C2”) + A2 + B
23      Part2 = C + a4
24
25      midi = Midi(1, tempo=90)
26      midi.seq_notes(Part1 + Part2, track=0)
27      midi.write(”midi/abstraction.mid”)
```

Finally, notice how we combine these motifs to form bigger elements. We combine *a*, *a1*, *a2*, and *a3* to form *A* (line 7). *A2* is just a transposition of *A*, and *C* is a bunch of *c* elements together. By abstracting compound elements, we can name them (for instance, *A*) and manipulate as an unit (for example, *A2*).

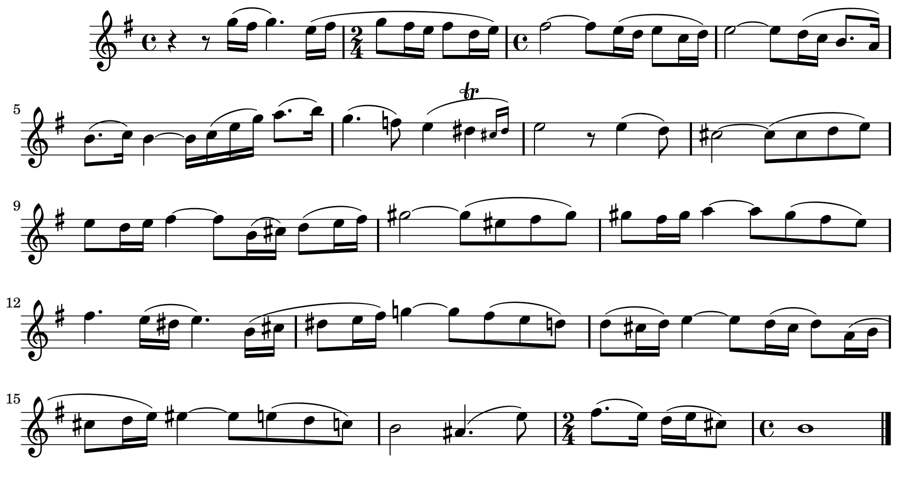
Exercise 16. Study the code to understand how the excerpt was formed. Make small changes and see how it sounds.

Exercise 17. Find the two wrong notes in the sheet music. If you did *Exercise 16* you may have noticed that two notes in the sheet music don’t match the notes generated by the code. The reason is that I actually wrote the music excerpt first, making the operations in my head. I decided to leave these two mistakes because I like them better, and to show that sometimes composers will make small deviations or mutations from the original plan, either by mistake or guided by their musical ear.

In the same way that throwing code in a class won't necessarily make the most useful and readable code, a music composition won't necessarily sound good just because we are using operations on music material. The music excerpt above sounds good (or at least decent if you're hypercritical ;-) because it's based on the same material (coherence), has some repetition (*A* and *A2*), variation (*A* and *C*), and the melody has a focal point (the Ab in measure 5) and a unique shape.

7.1 Example: Sergei Rachmaninoff, *Vocalize*

Exercise 18. The following example is the 1912 composition *Vocalize* by Sergei Rachmaninoff. Listen to it and try to recognize the repetitions and how the piece is organized in sections. Also notice how he basically uses only one or two musical ideas in the whole piece and a few operations we've seen so far. I won't show a code implementation for this song because it uses things we haven't seen (tonal transformations), and it'd be more complex. But feel free to try to implement the code to describe its compositional process.



7.2 Conclusion

In this chapter we have only scratched the surface of abstracting compound elements. This is a complex subject and the core of a music composition course. But hopefully you have seen how the music elements we saw in the previous chapters can be combined to make music. Check out our resources webpage to see book recommendations.

CHAPTER 8

Conclusion and Next Steps

In this book we have used programming to learn more about music. We saw the primitives of music and how to represent them with Python, how to combine primitives into larger units with operations, and how these primitives actually work from the inside.

It's hard to recommend things if I don't know your actual interest, but I'll try. Check this book's resources webpage for more suggestions and links:

- If you are interested in playing an instrument, try to incorporate this knowledge in your instrument practice. For instance, can you make operations like transposition and inversion in your instrument?
- If you are interested in learning more about mathematics and music, you may want to learn post-tonal theory. Straus' *Introduction to Post-Tonal Theory* is a good book. Also, Benson's *Music: a Mathematical Offering* is freely available.
- If you are interested in programming, check out Python libraries

such as music21 or Mingus.

- If you are interested in doing research involving music, the International Society for Music Information Retrieval Conferences have a lot of good stuff. They have all proceedings available at http://ismir2012.ismir.net.
- If you are interested in generating interesting sounds, Csound is not a Python package (although it can be scripted in Python), but you can have a lot of fun with it. I recommend Dodge's *Computer Music: Synthesis, Composition, and Performance* and Boulanger's *The Csound Book*.

CHAPTER 9

List of Exercises

- *Exercise 13*
- *Exercise 14*
- *Exercise 15*
- *Exercise 16*
- *Exercise 17*
- *Exercise 18*

CHAPTER 10

List of Tracks

- *Track 1*
- *Track 2*
- *Track 3*
- *Track 4*
- *Track 5*
- *Track 6*
- *Track 7*
- *Track 8*
- *Track 9*
- *Track 10*
- *Track 11*
- *Track 12*

- *Track 13*
- *Track 14*
- *Track 15*
- *Track 16*
- *Track 17*
- *Track 18*
- *Track 19*
- *Track 20*
- *Track 21*
- *Track 22*
- *Track 23*
- *Track 24*
- *Track 25*
- *Track 26*
- *Track 27*
- *Track 28*
- *Track 29*
- *Track 30*
- *Track 31*

CHAPTER 11

References

Brinkman, A. R. "A Binomial Representation of the Pitch Parameter for Computer Processing of Musical Data." Proceedings of the 1980 International Computer Music Conference. Ed. Herbert S. Howe. San Francisco: Computer Music Association, 1982. Print.

Cleonides. "Harmonic Introduction." *Strunk's Source Readings in Music History: Greek Views of Music*. Ed. Thomas J. Mathiesen. New York: W.W. Norton & Company, 1998. Print.

Drabkin, William. "Octave(i)." Grove Music Online.

Forte, Allen. *The Structure of Atonal Music*. New Haven: Yale University Press, 1973. Print.

Hewlett, Walter B. "A Base-40 Number-line Representation of Musical Pitch Notation." *Musikometrika* 4 (1992): 1-14. Print.

McNaught, W. G. "The History and Uses of the Sol-fa Syllables." Proceedings of the Musical Association 19th Sess. London: Royal Musical Association, 1893.