7
Strengthen Your Python Foundations

The code examples that are provided along with the chapters don't require you to master Python. However, they will assume that you previously obtained a working knowledge of at least the basics of Python scripting. It will also assume, in particular, that you know about data structures, such as lists and dictionaries, and how to make class objects work.

If you don't feel confident about the above mentioned subject or have minimal knowledge of the Python language, we suggest that before you start reading the book, you should take an online tutorial, such as the Code Academy course at `http://www.codecademy.com/en/tracks/python` or Google's Python class at `https://developers.google.com/edu/python/`. Both the courses are free, and in a matter of a few hours of study, they should provide you with all the building blocks that will ensure that you enjoy this book to the fullest.

We have also prepared a few notes, which are arranged in this brief but challenging bonus chapter, in order to highlight the importance and strengthen your knowledge of certain aspects of the Python language.

In this bonus chapter, you will learn the following:

- What you should know about Python to be an effective data scientist
- The best resources to learn Python by watching videos
- The best resources to learn Python by directly writing and testing code
- The best resources to learn Python by reading

Your learning list

Here are the basic Python structures that you need to learn to be as proficient as a data scientist. Leaving aside the real basics (numbers, arithmetic, strings, Booleans, variable assignments, and comparisons), the list is indeed short. We will briefly deal with it by touching upon only the most recurrent structures in data science projects. Remember that the topics are quite challenging, but they are necessary if you want to write effective code:

- Lists
- Dictionaries
- Classes, objects, and **Object-Oriented Programming** (**OOP**)
- Exceptions
- Iterators and generators
- Conditionals
- Comprehensions
- Functions

Take it as a refresher or a learning list, depending on your actual knowledge of the Python language. However, examine all the proposed examples because you will come across them again during the course of the book.

Lists

Lists are collections of elements. Elements can be integers, floats, strings, or generically, objects. Moreover, you can mix different types together. Besides, lists are more flexible than arrays since arrays allow only a single datatype.

To create a list, you can either use the square brackets or the `list()` constructor, as follows:

```
a_list = [1, 2.3, 'a', True]
an_empty_list = list()
```

The following are some handy methods that you can remember while working with lists:

- To access the *i*th element, use the `[]` notation.

  Remember that lists are indexed from 0 (zero), that is, the first element is in the position 0.

  ```
  a_list[1]
  # prints 2.3
  a_list[1] = 2.5
  # a_list is now [1, 2.5, 'a', True]
  ```

- You can slice lists by pointing out a starting and ending point (the ending point is not included in the resulting slice), as follows:

  ```
  a_list[1:3] # prints [2.3, 'a']
  ```

- You can slice with skips by using a colon-separated `start:end:skip` notation so that you can get an element every skip value, as follows:

  ```
  a_list[::2]
  # returns only odd elements: [1, 'a']
  a_list[::-1]
  # returns the reverse of the list: [True, 'a', 2.3, 1]
  ```

- To append an element at the end of the list, you can use `append()`, as follows:

  ```
  a_list.append(5L)
  # a_list is now [1, 2.5, 'a', True, 5L]
  ```

- To get the length of the list, use the `len()` function, as follows:

  ```
  len(a_list)
  # prints 5
  ```

- To delete an element, use the `del` statement followed by the element that you wish to remove, as follows:

  ```
  del a_list[0]
  # a_list is now [2.5, 'a', True, 5L]
  ```

- To concatenate two lists, use `+`, as follows:

  ```
  a_list += [1, 'b']
  # a_list is now [2.5, 'a', True, 5L, 1, 'b']
  ```

- You can unpack lists by assigning lists to a list (or simply a sequence) of variables instead of a single variable, as follows:

  ```
  a,b,c,d,e,f = [2.5, 'a', True, 5L, 1, 'b']
  # a now is 2.5, b is 'a' and so on
  ```

Remember that lists are mutable data structures; you can always append, remove, and modify elements. Immutable lists are called tuples and are denoted with round parentheses, `(` and `)`, instead of the square brackets as in the list, `[` and `]`, as follows:

```
tuple(a_list)
# prints (2.5, 'a', True, 5L, 1, 'b')
```

Dictionaries

Dictionaries are like hash tables—each key is associated to a value. Keys and values can belong to different kinds of datatypes. The only requisite for keys is that they should be hashable (that's a fairly complex concept; don't try to use a dictionary or a list as a key).

To create a dictionary, you can use the curly brackets, as follows:

```
b_dict = {1: 1, '2': '2', 3.0: 3.0}
```

The following are some handy methods that you can remember while working with dictionaries:

- To access the value indexed by the `k` key, use the `[]` notation, as follows:

  ```
  b_dict['2']
  # prints '2'
  b_dict['2'] = '2.0'
  # b_dict is now {1: 1, '2': '2.0', 3.0: 3.0}
  ```

- To insert or replace a value for a key, use the `[]` notation again, as follows:

  ```
  b_dict['a'] = 'a'
  # b_dict is now {3.0: 3.0, 1: 1, '2': '2.0', 'a': 'a'}
  ```

- To get the number of elements in the dictionary, use the `len()` function, as follows:

  ```
  len(b_dict)
  # prints 4
  ```

- To delete an element, use the `del` statement followed by the element that you wish to remove:

  ```
  del b_dict[3.0]
  # b_dict is now {1: 1, '2': '2.0', 'a': 'a'}
  ```

Remember that dictionaries, like lists, are mutable data structures. Also remember that if you try to access an element whose key doesn't exist, a `KeyError` exception will be raised:

```
b_dict['a_key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a_key'
```

The obvious solution to this is to always check first whether an element is in the dictionary:

```
if  'a_key' in b_dict:
    b_dict['a_key']
else:
    print "'a_key' is not present in the dictionary"
```

Otherwise, you can use the `.get` method. If the key is in the dictionary, it returns its value; otherwise, it returns `None`.

Defining functions

Functions are ensembles of instructions that usually receive specific inputs from you.

You can define them as one-liners, as follows:

```
def half(x) : return x/2.0
```

You can also define them as a set of many instructions in the following way:

```
import math
def sigmoid(x):
  try:
    return 1.0 / (1 + math.exp(-x))
  except:
    if x < 0:
      return 0.0
    else:
      return 1.0
```

To invoke a function, write the function name, followed by its parameters within the parenthesis:

```
half(10)
# prints 5.0
sigmoid(0)
# prints 0.5
```

Using functions, you normally ensemble repetitive procedures by formalizing their inputs and outputs without letting their calculation interfere in any way with the execution of the main program. In fact, unless you declare that a variable is a global one, all the variables you used inside your function will be disposed, and your main program will receive only what has been returned by the `return` command.

By the way, please be aware that if you pass a list to a function—only a list, this won't happen with variables—this will be modified, even if not returned, unless you copy it (see the `copy`, `deepcopy`, and `operator [:]` functions).

Why does this happen? This occurs because lists are in particular data structures that are referenced by an address and not by the entire object. So, when you pass a list to a function, you are just passing an address to the memory of your computer, and the function will operate on that address by modifying your actual list.

```
a_list = [1,2,3,4,5]
def modifier(L):
  L[0] = 0
def unmodifier(L):
  M = L[:]
  M[0] = 1
unmodifier(a_list)
print a_list # you still have the original list, [1, 2, 3, 4, 5]
modifier(a_list)
print a_list # your list have been modified: [0, 2, 3, 4, 5]
```

Classes, objects, and OOP

Classes are collections of methods and attributes. To create a class, use the `class` keyword. In the following example, we will create a class for an incrementer. The purpose of this object is to keep track of the value of an integer and eventually increase it by one:

```
class Incrementer(object):
    def __init__(self):
        print "Hello world, I'm the constructor"
        self._i = 0
```

Everything within the `def` indentation is a class method. In this case, the method named `__init__` sets the `i` internal variable to zero (it looks exactly like a function described in the previous chapter). Look carefully at the method's definition. Its argument is `self` (this is the object itself), and every internal variable access is made through `self`. Moreover, `__init__` is not just a method, it's the constructor (it's called when the object is created). In fact, when we build an `Incrementar` object, this method is automatically called, as follows:

```
i = Incrementer()
# prints "Hello world, I'm the constructor"
```

Now, let's create the `increment()` method, which increments the `i` internal counter and returns the status. Within the class definition, include the method, as follows:

```
    def increment(self):
        self._i += 1
        return self._i
```

Then, run the following code:

```
i = Incrementer()
print i.increment()
print i.increment()
print i.increment()
```

The preceding code results in the following output:

```
Hello world, I'm the constructor
1
2
3
```

Finally, let's see how to create methods that accept parameters. We will now create the `set_counter` method, which sets the `_i` internal variable.

Within the class definition, add the following code:

```
    def set_counter(self, counter):
        self._i = counter
```

Then, run the following code:

```
i = Incrementer()
i.set_counter(10)
print i.increment()
print i._i
```

The preceding code gives the following output:

```
Hello world, I'm the constructor
11
11
```

Note the last line of the preceding code where you access the internal variable. Remember that in Python, all the internal attributes of the objects are public by default, and they can be read, written, and changed externally.

Exceptions

Exceptions and errors are strongly correlated, but they are different things. An exception, for example, can be gracefully handled. Here are some examples of exceptions:

```
0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

len(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: len() takes exactly one argument (2 given)

pi * 2
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
```

In this example, three different exceptions have been raised (see the last line of each block). To handle exceptions, you can use a try/except block in the following way:

```
try:
    a = 10/0
except ZeroDivisionError:
    a = 0
```

You can use more than one `except` clause to handle more than one exception. You can eventually use a final "all-the-other" exception case handle. In this case, the structure is as follows:

```
try:
    <code which can raise more than one exception>
except KeyError:
    print "There is a KeyError error in the code"
except (TypeError, ZeroDivisionError):
    print "There is a TypeError or a ZeroDivisionError error in
    the code"
except:
    print "There is another error in the code"
```

Finally, it is important to mention that there is the final clause, which will be executed in all circumstances. It's very handy if you want to clean up the code (closing files, de-allocating resources, and so on). These are the things that should be done independently, regardless of whether an error has occurred or not. In this case, the code assumes the following shape:

```
try:
    <code that can raise exceptions>
except:
    <eventually more handlers for different exceptions>
finally:
    <clean-up code>
```

Iterators and generators

Looping through a list or a dictionary is very simple. Note that with dictionaries, the iteration is key-based, which is demonstrated in the following example:

```
for entry in ['alpha', 'bravo', 'charlie', 'delta']:
    print entry
# prints the content of the list, one entry for line

a_dict = {1: 'alpha', 2: 'bravo', 3: 'charlie', 4: 'delta'}
for key in a_dict:
    print key, a_dict[key]

# Prints:
# 1 alpha
# 2 bravo
# 3 charlie
# 4 delta
```

On the other hand, if you need to iterate through a sequence and generate objects on the fly, you can use a generator. A great advantage of doing this is that you don't have to create and store the complete sequence at the beginning. Instead, you build every object every time the generator is called. As a simple example, let's create a generator for a number sequence without storing the complete list in advance:

```
def incrementer():
    i = 0

    while i<5:
      yield i
      i +=1

for i in incrementer():
    print i

# Prints:
# 0
# 1
# 2
# 3
# 4
```

Conditionals

Conditionals are often used in data science since you can branch the program. The most frequently used one is the `if` statement. It works more or less as in other languages. Here's an example of this:

```
def is_positive(val):
  if val < 0:
      print "It is negative"
  elif val > 0:
      print "It is positive"
  else:
      print "It is exactly zero!"


is_positive(-1)
is_positive(1.5)
is_positive(0)


# Prints:
# It is negative
# It is positive
# It is exactly zero!
```

The first condition is checked with `if`. If there are any other conditions, they are defined with `elif` (this stands for else-if). Finally, the default behavior is handled by `else`.

Note that `elif` and `else` are not essentials.

Comprehensions for lists and dictionaries

Using comprehensions, lists and dictionaries are built as one-liners with the use of an iterator and a conditional when necessary:

```
a_list = [1,2,3,4,5]
another_list = ['a','b','c','d','e']
a_power_list = [value**2 for value in a_list]
# the resulting list is [1, 4, 9, 16, 25]
```

```
filter_even_numbers = [value**2 for value in a_list if value % 2
== 0]
# the resulting list is [4, 16]
a_dictionary = {key:value for value, key in zip(a_list,
another_list)}
# zip is a function that takes as input multiple lists of the same
length and iterates through each element having the same index at
the same time, so you can match the first elements of every lists
together, and so on.
# the resulting dictionary is {'a': 1, 'c': 3, 'b': 2, 'e': 5,
'd': 4}
```

Comprehensions are a fast way to filter and transform data that is present in any iterator.

Learn by watching, reading, and doing

What if the refresher courses and our learning list is not enough and you need more support to strengthen your knowledge of Python? We will recommend further resources that are available for free on the Web. By watching tutorial videos, you can try out complex and different examples and challenge yourself in a difficult task that requires you to interact with other data scientists and Python experts.

MOOCs

MOOCs have become increasingly popular in recent years, offering for free on their online platforms some of the best courses from the best universities and experts from around the world. You will find Python courses on Coursera (`https://www.coursera.org/`), Edx (`https://www.edx.org/`), and Udacity (`https://www.udacity.com`). Another great source is the MIT open courseware, which is easily accessible (`http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00sc-introduction-to-computer-science-and-programming-spring-2011/`). When you consult each of these sites, you may find different active courses on Python. Personally, we recommend a free, always available, and *do it at your own pace* course by Peter Norvig, the Director of Research at Google Inc. This course aims to take your knowledge of Python to a higher level of proficiency.

PyCon & PyData

The **Python Conference** (**PyCon**) is an annual convention organized at various locations around the world with the purpose of promoting the usage and diffusion of the Python language. During such conventions, tutorials, hands-on demonstrations, and training sessions are commonly held. You can check `http://www.pycon.org/` to know where and when the next PyCon will be held near you. If you cannot attend it, you can still perform a search on `www.youtube.com` because most of the interesting sessions are recorded and uploaded there. Attending and watching the real demonstration is a different thing anyway, so we warmly suggest you to attend such conventions because it is really worth it. Similarly, PyData, a community of Python developers and users devoted to data analysis, held many events around the world. You can check out `http://pydata.org/events/` for upcoming events (to go to and attend) and check whether there was anything that may have interested you in the past. As with PyCon, presentations are often available on YouTube.

Interactive IPython

Sometimes, you need some written explanations and the opportunity to test some sample code by yourself. IPython, an open tool like Python itself, offers you all of this via its notebooks—interactive web pages where you will find both explanations and example code that can be directly tested. We will devote explanations about IPython throughout the book; it is our data science workhorse for the running of Python scripts and the evaluation of their effects on the data that you work on.

The GitHub location of the IPython project offers a completely curated list of notebooks. You can check it out at: `https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks`. In particular, a section of the list is about *General Python Programming*, whereas another one is about *Statistics, Machine Learning and Data Science*, where you will find quite a lot of examples of Python scripts that you can take inspiration from in your learning.

Don't be shy, take a real challenge

If you want to do something that can take your Python coding ability to a different level, we suggest you to go and take a challenge on Kaggle. Kaggle (`www.kaggle.com`) is a platform for predictive modeling and analytic competitions, which applies the idea of competitive programming (participants try to program according to the provided specifications) in data science by proposing challenging data problems to participants and asking them to provide possible solutions that are evaluated on a test set. The results of the test set are partly public, partly private. The most interesting part for a Python learner is the opportunity to take part in a real problem with no obvious solution, which requires you to code something to even propose the simplest possible solutions to the problem (which we warmly suggest you to start first with). By doing so, the learner will come across interesting tutorials, beat-the-benchmark codes, helpful communities of data scientists, and some very smart solutions proposed by other data scientists or Kaggle itself in its blog, *no free hunch* (`http://blog.kaggle.com/`).

You may wonder how to find the right challenge for yourself. Just have a look at the actual, and past competitions at `http://www.kaggle.com/competitions` and look for every competition that has knowledge as a reward. You will be surprised to find an ideal stage to learn about how other data scientists code in Python, and you can immediately apply what you learn from this book.