Python Workshop for Beginners

Steven Lakin

Table of Contents

The Zen of Python: **2**
Introduction **3**
Installing Python **4**
Session 1
- **Section 1.1: Variables and Basic Operations** **5**
- **Section 1.2: Objects, Types, Attributes, and Methods** **6**
- **Section 1.3: Strings as a Data Structure** **8**
- **Section 1.4: Lists as a Data Structure** **12**
- **Section 1.5: Dictionaries as a Data Structure** **15**
- **Section 1.6: Reading From Files** **17**
- **Section 1.7: Rosalind Problem – Counting DNA Nucleotides** **19**

The Zen of Python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Introduction

With the advent of big data and the need for automation of repetitive tasks, basic programming skills are becoming a necessary skill for working in many fields. However, much of the instructional material on programming is intended to build a very solid foundation in programming basics. For our purposes, we can skip the details of rudimentary programming and take a more hands-on approach to basic scripting, since this is much of what we as non-computer scientists will be doing. We will be using the Python language, since it is an intuitive and powerful programming language.

Python, named after the Monty Python skits, was built with the intention of being easy to use, quick to learn, and syntactically fast; this is sometimes referred to in the documentation as being "Pythonic," based on the ideals of the language. Because of this, Python is what is called a "high-level" language; much of the clunky syntax of other languages (Java, C, R) is removed in python. There are no semi-colons at ends of lines, no brackets around portions of code, or the need to pre-define variables before use. You will find that this makes Python a fast language to program in, since we don't have to worry as much about typing and checking syntax. Much of the baseline "work" of lower-level languages has been built into Python, which allows you to access Python's intuitive structures and do your work as easily and fast as possible.

In this workshop, we will be focusing on applying Python to biological problems. Much of this "patchwork" programming for solving small problems is well-suited to short segments of code called scripts. A script is simply a file of code that does something. Scripts can be combined to make programs, packages, modules, and generally "software," all of which are generally the same thing with slightly different semantics in different languages. We will mostly be scripting in this class, though we will work toward making packages and learning the basics of building more advanced code structures.

The workshop will be divided into two segments: the first will be a lecture on a programming topic, and the second will be applying those concepts to miniature problems on Rosalind, named after Rosalind Franklin, whose work in X-ray crystallography contributed significantly to the discovery of DNA structure. These problems are bioinformatics related, which is a field that combines biology, computer science, mathematics, and statistics to solve biological problems involving large data sets. You will need to make an account on Rosalind to track your progress.

Installing Python
There are two primary versions of Python: Python2.7 and Python 3.4. For the purposes of learning Python and having the most backward compatibility, I recommend that you install Python2.7. That being said, if you are using 3.4, we can work with that!

Windows and Mac
On Windows and Mac, you can download executable files for installation from
https://www.python.org/downloads/

Simply follow the prompts for installing Python, and you may find it useful to check the box "Add Python to PATH" for reasons we will discuss later.

Linux
On Ubuntu and Debian flavors of Linux, you can obtain Python by apt-get:

```
http://askubuntu.com/questions/101591/how-do-i-install-python-2-7-2-on-ubuntu
```

Text editors and IDEs

At the very least, you will need to have a basic raw text editor (not Word). You can use Notepad on windows, the basic text editor on Mac, or you can download a simple editor like Editra.

Alternatively, if you want more functionality (and you will eventually), you can look into one of the Python IDE's. IDE stands for Interactive Development Environment, and it will allow you to write scripts, run the scripts in the terminal, and access help documentation all in the same environment. A good example of an IDE is RStudio for the R language.

There are many IDEs for Python, and no one is the "best," so you will have to find your preference. An IDE with a lot of functionality is PyCharm, but it is a larger program and can be confusing when you're starting out. For now, if you find these to be too confusing, then stick with downloading a program like Editra and working in that.

Session 1: Variables, Data Structures/Types, Operations, and I/O

Section 1.1 - Variables and Basic Operations

The Python terminal is an interface to Python's functionality. The terminal "prompt" is denoted by three right wedges:

```
>>>
```

When you see this prompt, the terminal is telling you that you can enter commands. When this prompt is not visible, then Python is "working" on a command you have entered previously. The terminal is what is known as an "interactive" terminal. You can use it like you might use a calculator for basic operations:

```
>>> 2+2
4
>>> 2*3
6
```

While this is useful to us, we are more interested in working with information stored in the Python environment. How do we store data? By assigning it to a variable:

```
>>> a = 2
>>> b = 3
>>> a*b
6
>>> c = a + b
>>> c
5
```

Let's try division:

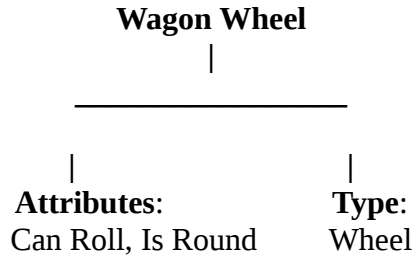
```
>>> a/b
0
```

Wait a tick, that's not right... This is because we are working with integers and to get fractions, we need to tell Python to give us a value of the right **type**:

```
>>> float(a)/b
0.6666666666666666
```

Section 1.2 - Objects: Types, Attributes, and Methods

Let's talk about types. It is best (and it will help us later) to think of data that we put into Python as **objects**. Objects, philosophically speaking, have certain attributes, and we can envision that there are objects of various types. Consider a wagon wheel: it has **attributes**, such as that it is round (by definition) or that it can roll, and it is of a certain **type**, which is that it is a specific kind of wheel.



Types describe a general class that your object falls into, and **attributes** describe what you can do to that specific object. A third term, **methods**, describe what that object can do for you. Let's use a concrete Python example that we have already encountered: the **Int** type, short for Integer.

The type() command will tell us what type our object is:

```
>>> type(c)
<type 'int'>
```

The dir() command will tell us what is in our object, which are it's **attributes** and **methods**:

```
>>> dir(c)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__', '__div__',
'__divmod__', '__doc__', '__float__', '__floordiv__', '__format__', '__getattribute__', '__getnewargs__',
'__hash__', '__hex__', '__index__', '__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mod__',
'__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

That's a lot of attributes and methods! What do they mean? They tell us what kinds of operations we can or can't do to the object. The methods also tell us what the object is capable of doing, such as telling us how big it is in computer-language-size (bits) with the method in bold:

```
>>> c.bit_length()
3
```

There are many types, but the common ones that we will be working with are **string**, **int**, **float**, and the data structure types **list** and **dictionary**. These are common because they are **mutable**, which means

we can manipulate them in different ways. Let's see a few examples of these basic types and how we can work with them:

Section 1.3 – Strings as a Data Structure:

Strings are words made up of letters. Strings are defined with quotes, either single or double quotes:

```
>>> dna = 'AGCT'
>>> type(dna)
<type 'str'>
>>> dna
'AGCT'
```

is the same as doing it with double quotes:

```
>>> dna = "AGCT"
>>> type(dna)
<type 'str'>
>>> dna
'AGCT'
```

But let's say we are interested in specific nucleotides in the DNA string. We can access each individual letter by its position in the string, starting with 0 and going to 3:

```
>>> dna[0]
'A'
>>> dna[1]
'G'
>>> dna[2]
'C'
>>> dna[3]
'T'
```

Or certain **slices** of the DNA string using colons:

```
>>> dna[0:2]
'AG'
>>> dna[2:4]
'CT'
```

Python uses **zero indexing**, which means the indices start at zero. Visualize it like this for our DNA string:

$$_0A_1G_2C_3T_4$$

So when we slice from 0 to 2, we get out 'AG'. When we slice from 2 to 4, we get out 'CT'. If we want a single element, we use the index on the left side of it, so the index of A is 0. We're not limited to

consecutive slices either, let's say we want every other element:

```
>>> dna[::2]
'AC'
```

So the notation looks like this in general:

StringObject[start:stop:by]

Where “by” is equivalent to “every x letters”. We can even reverse the direction of the string:

```
>>> dna[::-1]
'TCGA'
```

We can also use negative indices to refer to the end of the string, where 0 is the very beginning or end of the string, -1 is the last letter in the string, -2 is the second to last letter, and so on:

```
>>> dna[-1]
'T'
>>> dna[-2]
'C'
```

Note that slicing with negative indices doesn't work as well. We'll talk about alternative strategies for slicing the end of strings later.

Let's say we have new information about our DNA and want to add it on. Remember that strings are mutable, so we can do this with simple addition, or **concatenation**:

```
>>> moreDna = "TT"
>>> longDna = dna + moreDna
>>> longDna
'AGCTTT'
```

If we want to remove specific elements of our DNA string, we can use the replace() **method**:

```
>>> longDna.replace("GC","")
'ATTT'
```

Note that we replaced GC with nothing (empty quotes), which is the equivalent of removing it. Methods, in general, are used like this:

Object.method(arguments)

Where the arguments are comma separated and depend on which method you're using. Methods are defined by the type of object you're working with, so for instance, you could use the replace() method on any string, because the type of object we are currently discussing is the string type. If you ever have a question about what methods there are or what arguments are valid, Google has all the answers. It is often faster to Google it than to try to look it up in the Python documentation.

Let's say we have an int (integer) object and want to turn it into a string object. This is called **coercing** one type into another. Not all types can be coerced into other types, but string is a broad category, so many other types can be coerced into strings:

```
>>> c
5
>>> type(c)>>> counts[0]
1
>>> counts[1]
2
>>> counts[2]
3
>>> counts[0:3]
[1, 2, 3]
>>> counts[-1]
5
>>>
<type 'int'>
>>> c_string = str(c)
>>> c_string
'5'
>>> type(c_string)
<type 'str'>
```

Now our integer is a string. This will be useful for some applications later on. This is about all I want to discuss about strings at this point, but I will leave here some useful operations with strings for your reference.

Useful String Manipulations:

```
>>> exampleString = "AGCTTTTCA"
```

Length of a string:

```
>>> len(exampleString)
9
```

Count of a non-overlapping pattern in a string:

```
>>> exampleString.count('A')
2
```

Find the first occurrence of a pattern in a string:

```
>>> exampleString.find('T')
3
```

Split a string at a defined pattern, remove the pattern, return a list:

```
>>> exampleString.split("GCT")
['A', 'TTTCA']
```

Join multiple strings with a separator (note that this works on lists of strings):

```
>>> splitString = exampleString.split("GCT")
>>> splitString
['A', 'TTTCA']
>>> "_".join(splitString)
'A_TTTCA'
```

Remove leading or trailing white space (including line endings):

```
>>> whiteSpace = "I am a sentence with a line ending\n"
>>> whiteSpace
'I am a sentence with a line ending\n'
>>> whiteSpace.strip()
'I am a sentence with a line ending'
```

Section 1.4 – Lists as a Data Structure:

So strings are pretty nifty, but what if we want to store discrete elements in a structure? Lists are both famous and notorious for this. They are great for storing small data, but they can be inefficient computationally if you're changing them frequently with large data sets (by large, I mean anywhere above half a million manipulations or so will be noticeable).

For those interested, here is a quick comparison for generating lists versus arrays (which we will discuss in a later session). Lists are quite slower:

```
Arrays:
$ python -m timeit "x=(1,2,3,4,5,6,7,8)"
10000000 loops, best of 3: 0.0388 user sec per loop

Lists:
$ python -m timeit "x=[1,2,3,4,5,6,7,8]"
1000000 loops, best of 3: 0.363 user sec per loop
```

However, for our introduction to Python, lists will be a key piece of our code because they are easy to use. So let's take a closer look at them.

Lists are defined by square brackets, and its elements can be other objects, such as strings, ints, other lists, etc. Here, we will use integers:

```
>>> counts = [1,2,3,4,5]
>>> counts
[1, 2, 3, 4, 5]
```

We can access lists the same way we can strings:

```
>>> counts[0]
1
>>> counts[1]
2
>>> counts[2]
3
>>> counts[0:3]
[1, 2, 3]
>>> counts[-1]
5
```

However, if we want to add elements to a list, we need to use the append() method:

```
>>> counts
[1, 2, 3, 4, 5]
>>> counts.append(6)
>>> counts
[1, 2, 3, 4, 5, 6]
```

We can get the length of the list (or of any dimensional object) with len():

```
>>> len(counts)
6
```

Most of what was mentioned in the string section applies here as well, so I will just give examples of list operations here that might be useful for reference.

Useful List Manipulations:

Insert a value into a list at a given position (first argument is the index, second is the value):

```
>>> counts
[1, 2, 3, 4, 5, 6]
>>> counts.insert(7,2)
>>> counts
[1, 2, 3, 4, 5, 6, 2]
>>> counts.insert(3,10)
>>> counts
[1, 2, 3, 10, 4, 5, 6, 2]
```

Remove the first occurrence of an element from a list:

```
>>> counts
[1, 2, 3, 10, 4, 5, 6, 2]
>>> counts.remove(10)
>>> counts
[1, 2, 3, 4, 5, 6, 2]
```

Reverse a list:

```
>>> counts
[1, 2, 3, 4, 5, 6, 2]
>>> counts.reverse()
>>> counts
[2, 6, 5, 4, 3, 2, 1]
```

Sort a list:

```
>>> counts.sort()
>>> counts
[1, 2, 2, 3, 4, 5, 6]
```

Count the occurrence of elements in a list:

```
>>> counts.count(2)
2
>>> counts.count(4)
1
```

Section 1.5 – Dictionaries as a Data Structure:

Dictionaries are a mapping of one value to another, such that they are linked. We refer to the index as a **key**, that has an associated **value**. Keys are unique within the dictionary; you cannot have repeated keys. However, you can have repeated values. This makes dictionaries great for storing associations for things like counting, translating, and storing associations in data sets. Let's take a look at how they work:

Dictionaries are defined by braces (curly brackets) where the key is mapped to its value with a colon:

```
>>> rnaTranslate = {"UUU":"F", "UUC":"F", "UUA":"L"}
>>> rnaTranslate
{'UUU': 'F', 'UUA': 'L', 'UUC': 'F'}
```

Here, we have made a dictionary that translates some of the RNA codons into their respective amino acids. The **keys** are the codons of RNA nucleotides, and the **values** are the single amino acid letter. Notice that we have a degenerate genetic code, which means we have more than one codon that maps to the same amino acid. Therefore, we must use the codons as the **keys**, because **keys must be unique**. We couldn't reverse the keys and values in this dictionary because the letter F repeats. **Values do not have to be unique**. So the amino acids are acceptable as the **values**.

Dictionaries have useful features beyond what a list or string can provide. For instance, we can get a list of the keys like so:

```
>>> rnaTranslate.keys()
['UUU', 'UUA', 'UUC']
```

And the values:

```
>>> rnaTranslate.values()
['F', 'L', 'F']
```

You can get the values out by using the keys as indices:

```
>>> rnaTranslate['UUU']
'F'
```

Now let's say I have a list of RNA codons that need to be translated:

```
>>> codons = ['UUU', 'UUU', 'UUU', 'UUA', 'UUU', 'UUC']
```

Let's translate them into amino acids:

```
>>> [rnaTranslate[x] for x in codons]
['F', 'F', 'F', 'L', 'F', 'F']
```

Pretty cool right? Try it on your terminal. This is called **list comprehension** and is part of what makes Python intuitive as a language. We will get into loops and flow in a later session, but this is one very Pythonic way to avoid using loops, which we will see can get confusing. I don't expect everyone to understand the above syntax yet, but I thought I'd demonstrate how useful associative structures like dictionaries can be. (Plus it's one of our Rosalind problems in a later session).

Here are some other fun things you can do with dictionaries.

Useful Dictionary Features

```
>>> exampleDict = {'A': 20, 'G': 120, 'C': 8, 'T': 11}
```

Sort a dictionary by its keys:

```
>>> sorted(exampleDict)
['A', 'C', 'G', 'T']
```

Get the key-value pairs out in a nested list:

```
>>> [[key,value] for key,value in exampleDict.items()]
[['A', 20], ['C', 8], ['T', 11], ['G', 120]]
```

Put them back into the dictionary:

```
>>> listData = [[key,value] for key,value in exampleDict.items()]
>>> listData
[['A', 20], ['C', 8], ['T', 11], ['G', 120]]
>>> { key:value for key,value in listData }
{'A': 20, 'C': 8, 'T': 11, 'G': 120}
```

Delete an entry in a dictionary:

```
>>> del exampleDict['A']
>>> exampleDict
{'C': 8, 'T': 11, 'G': 120}
```

Section 1.6 – Reading from Files (Input/Output, AKA I/O)

Much of what we will be doing will be working with data stored in files. Python can read data from files and store it in the environment so that you can manipulate it and then possibly write it to a new file. You can do this for many thousands of files with many thousands of line of data, which is part of what makes programming so useful. Here, I am going to introduce reading files only, since we won't need writing for a few sessions.

In Python, we explicitly open a file and assign that open file to a variable. I am going to use an example of a file on my computer, but you would substitute the location of my file with the location of yours. You can get the location of your file by opening your file browser window and clicking in the bar where it shows what folder you are in. That is called the **filepath**.

```
>>> openFile = open(r'/home/lakinsm/Documents/HelloWorld.txt', 'r')
>>> openFile
<open file '/home/lakinsm/Documents/HelloWorld.txt', mode 'r' at 0x7efd8cbc34b0>
```

You'll notice there is a lowercase “r” (in boldface) at the beginning of the filepath string. This is a special kind of string, called **raw string** or **string literal**. I'm not going to go into what this means for now, but I put it in there for the benefit of Windows people, because you need to put the “r” there for your filepaths to work. Windows filepaths have backslashes instead of forward slashes, which makes them a little harder to work with in Python. We can solve this problem by using the raw string. So if you are on Windows, your filepath might look liks this:

```
>>>openFile = open(r'C:\Documents\HelloWorld.txt', 'r')
```

So what do we have now? The variable openFile is telling Python where our file is on the computer and what we want to do with it. Mode “r” stands for **read**. We could have use “rw” instead of “r” and it would mean **read and write**. Though this is fine for now, we try to explicitly state what we want to do to the file so no accidents happen where we write over the file's contents by mistake.

So now how do we get the data in? We use the read() method:

```
>>> data = openFile.read()
>>> data
'Hello World!\n\n'
```

Great! But what are these \n characters? These are **newline** characters. If you're working in Windows, you might have \r\n instead. These are what tell your text editor programs to begin a new line. However, in Python, we don't care about them per se, since we want to work with the data in a more useful form. So, we can easily get rid of them with the method we learned in the string section:

```
>>> cleanData = data.strip()
>>> cleanData
'Hello World!'
```

And there you have it: you've just performed your first input and data cleaning operation in Python.

However, be careful with opening files like this. Unless you explicitly close the file, it will remain open. We don't want that, because it consumes your computer's RAM when it doesn't need to. So let's close the file:

```
>>> openFile.close()
```

Now we have the data in Python and the file is closed. You can now manipulate the data to your heart's desire and not worry about the file being open. Yet, sometimes we can forget to close files, so from this moment on, I'm going to use a more Pythonic way to open files, get the data, and close them:

```
>>> with open(r'/home/lakinsm/Documents/HelloWorld.txt', 'r') as openFile:
        cleanData = openFile.read().strip()
        Do Something With The Data
```

Note the indentation here; in Python, this is vital to get correct. I will discuss why this is in the next session, but for now remember that indentation is Python's way of knowing what to do in what order. Let's consider what we've done though. We have opened the file as openFile, imported the data into cleanData, and we (hypothetically) did something with it. The “with” statement make it so that when we are done doing whatever it is we want to do, the file will automatically be closed. This is a much cleaner and more Pythonic way of handling file reading.

Section 1.7 – Rosalind Problem 1: Counting DNA Nucleotides

Location: http://rosalind.info/problems/dna/

Problem:
A string is simply an ordered collection of symbols selected from some alphabet and formed into a word; the length of a string is the number of symbols that it contains.
An example of a length 21 DNA string (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset:
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTG
ATAGCAGC

Sample Output:
20 12 17 21

This Rosalind problem is about counting occurrences of a pattern in a string. Use the methods we discussed in the string section to output the variables.

You can get the values out of the variables all at once with print():

```
>>> A = 20
>>> C = 12
>>> G = 17
>>> T = 21
>>> print(A,C,G,T)
(20, 12, 17, 21)
```