Introduction to Python Programming and Project Design

Steven Lakin, DVM

2018 May 14

Contents

1 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!

2 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.

3 Obtaining Help

Every programmer needs a reference source for learning new skills in a language or to troubleshoot problems encountered during the programming process. There are many sources of this information, including physical textbooks or manuals. However, the vast majority of programmers these days utilize online resources that have been curated and developed by the community. Probably the most popular and frequently utilized online resource is Stack Overflow, which is part of the popular Stack Exchange system of websites for community-based question and answer forums. I would recommend Googling stack exchange plus whatever keywords or error codes that you have in order to determine the best path forward in your code. Very frequently, someone has encountered the same question or problam as you, and that question will already have been answered on Stack Overflow. However, if you do ever encounter a novel problem (it has only happened to me twice in my entire programming career), then you can submit a new problem on Stack Overflow and hopefully get the problem solved. Alternatively, the online Python manual often has answers about the language itself.

4 Python Fundamentals

4.1 Installing Python

Python Versions
There are two main distributions of Python: Python 2 and Python 3. These days, virtually every application should be compliant with Python 3, so it's recommended that you download the most recent version of Python 3. The difference between the two is minimial from a basic user perspective, however some of the "grammar" or syntax differs between them. This workshop will be conducted as if everyone has Python 3.

Python Download
You can download Python for Windows and MacOS by visiting the Python download page. Linux users can download Python through their appropriate package manager. I will assume if you're using Linux that you know the basics of the operating system.

Integrated Development Environments (IDEs)
When we get to the point where we are writing scripts, you may find it helpful to download specialized software called an IDE, which is essentially a text editor that knows the Python syntax, can highlight regions of code, helps with code completion, and can help debug programs. There are a wide array of IDEs for Python, some more feature-complete than others. I prefer to use a more complete IDE called PyCharm (Jetbrains), however some people prefer lighter IDEs, and others prefer to use a standard text editor like NotePad that has no coding-specific features. You can research a few of these IDEs and decide what is the right fit for your coding experience.

Python Packages
Python has both core functionality and the ability to be extended for other purposes through the development of packages. Packages are modules that perform a specific function, such as helping to scrape websites, download files, multiply matrices, and visualize data through graphing/plotting. You can obtain these packages through Python's package manager called Pip. A basic tutorial on how to use Pip to obtain packages can be found in the Python user manual.

4.2 Variables and Basic Operations

As a Python programmer, you'll encounter two situations commonly while coding: testing small bits of code in real time, and assembling larger pieces of code into a program or script. For testing code in real time, you'll use the **Python Interpreter**. To get to the Python Interpreter, double click on the Python icon or open a command-line terminal and type python in lower-case. The interpreter will have three greater-than signs as its prompt, like so:

```
>>>
```

The Python Interpreter has all of the functionality that we will be using later, however it is difficult to program larger pieces of code in the Interpreter, so we will be switching to scripting later on. The Interpreter can perform basic arithmetic such as a calculator would do; simply type the equation and press enter:

```
>>> 3 - 5
-2
>>> 3 + 5
8
>>> 3 * 5
15
>>> 3 / 2
1.5
>>> 3e5
300000.0
```

Notice that all input either starts with the three greater-than signs or three periods, while all output simply is on a blank newline. While performing basic arithmetic operations is a nice feature of Python, we could simply use a calculator for this. Python's functionality and power comes from being able to store values in **variables**:

```
>>> a = 3
>>> b = 5
>>> a + b
8
>>> a * b
15
>>> a * 5
15
```

For those interested in proper programming terminology, we refer to variables as being a **left-hand values** or l-values, since it is on the left hand side of the equation, while **right-hand values** or r-values represent the values that the variable will take. Remember that right-hand values **get assigned to** left-hand values; you cannot interchange the two.

In Python, we are not limited to numbers. Python, being a simple and elegant language, has only a few **types** of values. We have already encountered the **Integer** type and the **Float** type: integers referring to the mathematical definition of whole real numbers, and floats referring to decimal-point real numbers. A commonly-used third type is the **String** type, which holds words. Less commonly used but important is the **Boolean** type, which indicates one of two values: true or false (1 and 0, respectively in binary). Python often will automatically convert between types depending on what you need to do:

```
>>> a = 3
>>> b = "1"
>>> c = "String Example"
>>> print(a, b)
```

```
3 1
>>> c + b
'String Example1'
```

However, notice that Python will not automatically convert strings into integers or floats; to do that you'll have to **coerce** the value that you know complies with the new type into that type, like so:

```
>>> a = "1"
>>> b = 3
>>> a + str(b)
'13'
>>> int(a) + b
4
>>> float(a) + b
4.0
>>> bool(a)
True
```

Notice that the addition sign has different functionality depending on the type for which it is used: numbers are added together, while strings are **concatenated** in the order of which they were added. Other functionality that are built into Python include the native **functions** that you can use for common operations like printing text to the terminal:

```
>>> myvariable = "Hello World"
>>> print(myvariable)
'Hello World'
```

Functions take variables or values as input and are used with parentheses as above. Next, we will learn how to create our own functions as well, which is the foundation of programming.

4.3 Functions

Functions are the building blocks of programming; each function should have a single purpose, be specific, and have an easily-interpretable and appropriate name. The parts of a function include the **name**, the **arguments**, the **function body** where the code resides, and an optional **return value**. Functions in Python are **defined** like so:

```
>>> def function_name(argument1, argument2):
...     print(argument1)
...     print(argument2)
...     return argument1 + argument2
...
```

The above code defines the function called $function\_name$, which takes in two arguments, "argument1" and "argument2." It first prints each argument on a separate line, then returns their sum. By default, return values get printed to the Interpreter if they are not used. Alternatively, we can treat return values as right-hand values and assign them to another variable:

```
>>> function_name(1, 2)
1
2
3
>>> newvalue = function_name(1, 2)
1
2
>>> print(newvalue)
3
```

Notice that in our function definition, the function body (including the return statement) are indented; in Python, all code that falls into a **code block** such as a function needs to be indented the same number of spaces or tabs. The debate between the use of spaces or tabs is semantic but hotly debated amongst Python programmers. I personally prefer tabs, as they take fewer keystrokes, however you can use what you wish as long as you are consistent. Each level of indentation indicates another **nested** code block. We will clarify this in the following sections once we have the ability to nest code blocks.

Functions can take two kinds of arguments: **positional** and **optional** arguments. Positional arguments are mandatory when using the function and are always placed before optional arguments. Optional arguments can be placed in any order and have a default value explicitly specified in the function definition. There are various reasons to use positional versus optional arguments; perhaps your function will not work without a certain number of arguments, like the function we made above. Alternatively, if you want to specify a default value but still allow other values, you would use an optional argument. Here is a function definition that includes both positional and optional arguments:

```
>>> def add_and_multiply(summand1, summand2, coefficient=1):
...     return coefficient * (summand1 + summand2)
...

>>> add_and_multiply(2, 3)
5
>>> add_and_multiply(2, 3, 3)
15
>>> add_and_multiply(2, 3, coefficient=3)
15
>>> add_and_multiply(summand1=2, summand2=3, coefficient=3)
15
```

Arguments can be called by name or by position, however if you are striving to have legible code, it usually pays off to explicitly spell out the arguments so others (or you several years later) can read the code and understand your intentions.

With functions and variables, you have the most basic use cases of Python covered and can begin performing operations on input values. However, often times we want to store larger amounts of data than single values, and often those values need to be stored in a way that relates them to one another or orders them in a certain way. To do this, we will need to learn how to use the **data structures** that are inherently available in Python. However, we will first practice what we have learned so far on one of Rosalind's introductory bioinformatics problems: counting nucleotides.

Rosalind Problem 1: Counting Nucleotides
Click here to visit this problem on Rosalind

Often times we want to obtain information from string-based data, such as words, paragraphs, or in this case nucleotides from a DNA sequence. In this problem, you will use functions inherent to the **String type** to count the number of adenine, cytosine, guanine, and thymine nucleotides in a DNA string of variable length. To do this, make use of the **count** function that is inherent to the String type, and define a function that prints out the number of ACGT nucleotides in that order:

```
>>> dna = "ACGTGTGTGCCCGTGA"
>>> dna.count("A")
2
>>> dna.count("C")
4
```

4.4 Data Structures