The Python Pilgrim- Part One: The Theoretical Minimum

Tachibana Laboratories

"Let me see: four times five is twelve, and four times six is thirteen, and four times seven is -- oh dear! I shall never get to twenty at that rate!" —Lewis Carroll, Alice's Adventures in Wonderland

Foreword

Welcome to the beginning of a new journey. A new world can feel like a nonsensical and funny place where all of the rules you know are challenged and common sense is distorted, but as you will come to learn, this is not necessarily true (and were it true we may have discovered something quite brilliant too, much like Ernest Rutherford and the Geiger–Marsden experiment, or Thomas Young performing the double-slit experiment). With the right map you will translate the logic you know into the logic you need to learn. You will come to discover that you had the answers inside of you from the beginning. The objective of this series is to make something daunting, easy. A lot of guides and texts will address this topic with a lot more rigour and it is not the author's goal to replace them, it is however the author's goal to address the things that beginner programmers find confusing. It is also not explicitly the author's goal to rigorously cover "idiomatic Python" as this topic has been covered ad nauseum elsewhere, however later items in the series may address this. With these confusions resolved it should be quite simple for the reader to fill in the blanks themselves with their own dilligent research. To aid this step of the process I will include within this guide resources and tips as to how to fill in these gaps. Well then, shall we begin?

Glossary of common terms and syntax

: An inline comment operator, put this before text you want to comment but don't want the Python interpreter to interact with

" " : Anything these quotation marks enclose is defined as a string

\ : Division, numerator on left, denominator on right. Version specific behaviour is document in the Versions section

\\ : Integer division, divides and then rounds down to nearest one's place. Integer dividing a float will still return a float but it will be rounded

* : Multiplication, behaves the same on integers and floats. An integer can be multiplied by a float and this will give a float

+ : Addition

- : Subtraction

<,>, <=, >= : Arithmetic comparators. Less than, greater than, less than or equal to, greater than or equal to

% : Modulo, divides and then gives the remainder, numerator on left, denominator on right. If numerator < denominator then x%y=x. Works on floats

object : Object has a special meaning in object oriented programming relating to having attributes, methods, being instanced and having the mechanism of inheritance. However more generally an object can be any variable, function/ method or data structure. One sign of an object is that it can be assigned to a variable or passed as an argument

mutable/ immutable : a mutable object is one that can be changed as such as a list as it can have its elements deleted, modified and added. An immutable object is one that cannot be changed, like an integer or a string. When performing arithmetic on an integer the integer is not modified, instead a new integer is returned as the result. Immutable objects exist because only they can be hashed, and have greater runtime performance for certain operations

return : Return is a statement that breaks out of a function and passes an object back to the calling function, which can be assigned to a variable, this is explained in greater detail in the Functions section

list = [a, b, c] : Initialises a new list. Can contain greater than or equal to zero comma space elements. List can be initialised as empty by leaving the square brackets empty. List can contain any Python compatible typed elements, and data structures

list[x] : Accesses element of a list by index where x is an integer. Can be used to get or set an element

list.append(x) : Adds element x to list. A full list of list functions can be found **here**

tuple = (a, b, c) : A tuple is an immutable data structure that stores comma separated objects. Elements can be accessed by index in the same way as a list, but cannot be altered. Python syntax allows a tuple to be assigned without parentheses

dictionary = {key:value, key:value} : A dictionary is a data structure that consists of comma separated key:value pairs. A key can be any immutable object and a value can be any object

dictionary.get(key) : Returns the value associated with the key object supplied

The Python interpreter

Document structure

Each Python script follows a simple template. I will begin with presenting a very simple script that incorporates all essential elements you will encouter before learning about classes.

```
#!/usr/bin/env

import module


def function():
''' function prints the string "Hello, world!" '''
        print("Hello, world!")


function()
```

Let's analyse this script piece by piece.

Shebang

```
#!/usr/bin/env
```

This is referred to as a shebang. In UNIX like operating systems when a Python script is made executable, and with certain software like the Python launcher for Windows, this line of code defines the location of the default interpreter. Defining an interpreter when running a script from and interactive development environment, or terminal will override this variable.

Import

```
import module
```

The import statement is used to make the features of external modules available to your script. "Module" is not the name of a module, just a placeholder for a module's name.

Function definition

```
def function():
''' function prints the string "Hello, world!" '''
        print("Hello, world!")
```

This is called a function definition. For now let's just highlight that it consists of the def statement, followed by the function's name with parentheses at the end in which arguments in the form of objects can be passed for the purpose of being made available to the code it contains. Inside of the function definition is the code that will run if the function is *called*. This function object will be discussed in much greater length in the Function section.

Docstring

```
'''function prints the string "Hello, world!"'''
```

This is called a Docstring. It is used to document elements of your code, in the case it is for the function() function. A Docstring isn't necessary for your code to run but it is a good practice to document your functions so that it can be easily understood. In python 3.7, if a Python script is opened from within the Python environment (by calling the single argument 'python' from the command line) the following functionality is available.

```
>>>print(function.__doc__)
function prints the string "Hello, world!"
```

This same print function can be made available in Python 2.7 with the following import statement:

```
from __future__ import print_function
```

Function call

```
function()
```

Stating the function name on its own followed by the closed parentheses "()" tells the interpreter to run that function, also known as "calling" the function. Functions can be called with parameters inside of the parentheses, and the output of the function can be assigned to a variable on the same line that it is called, but don't worry about it for now if you don't understand what these things mean, that will be explained later. For now just remember that this is how you run a function.

Variables

In very high level terms a variable is like a container that you can store data inside of. A variable can only store one piece of data (in the case of python an "object") at a time, and that object can only have one type. This, however, does not mean that you cannot store data *structures* inside of variables that can contain multiple pieces of data with multiple types. I will go on to explain this shortly. If you are feeling adventurous consider the following (the following is **OPTIONAL** and not necessary to understanding Python). More accurately a variable is a section of your computer's volatile memory (RAM) (actually with paged memory variables may be allocated onto your hard disk before being swapped back into RAM for execution but this is *way* beyond the scope of this guide) that has a size in bits allocated based upon the size of the type of data it will contain. When you declare a variable you tell the computer to allocate that memory, and to associate the variable with it such that the variable "points" to those memory addresses. This means that if you have a data structure like a "list" inside of a variable, those memory addresses will contain contiguous variables itself (pointers) that point to that data structure's elements in other memory addresses. This is how Python lists can have multiple data types as their elements. A list's memory addresses are known as a *dynamic* array which means more memory addresses are added to the variable if the list grows.

Assignment

Python follows a strict rule for assigning a value to a variable. An assignment always follows a right to left assignment order where the variable is on the left, and that object being assigned to the variable is on the right. It is implemented as follows:

```
#not real code, just an illustration of assignment
variable <- data

#real code assigning the integer 1 to the variable x
x = 1

#this can be proven with the print() function
>>>x = 1
>>>print(x)
1

#it is exactly the same for different object types
list = [1, 2, 3]

true_or_false = True

state_capital_dictionary = {QLD: Brisbane, NSW: Sydney}
```

There is also a concept in Python called multiple assignment, or tuple unpacking, it is where an element of a tuple is assigned to an variable in the corresponding position in a comma separated sequence. It is performed as follows:

```
#assigns x = 1, y = 2
x, y = (1, 2)

#the same thing can be done without parentheses as they are not necessary for a tuple
x, y = 1, 2

#sometimes you will have a function that returns a tuple, the process is the same
x, y = tuple_function()
```

Assignment can also be done to indices of certain mutable objects like lists. The following will demonstrate how a variable can be assigned to a particular index of a list.

```
list = ["a", "b"]
```

Types

Boolean logic

Data structures

Control flow

The dot operator

Statements

Functions

Formatting

Indentation blocks

In the Python programming language, most functionality can be broken down into a pair of entities: (a) one dictates the conditions of the code's execution, (b) and the other is the code to be executed. This grouping is called a *block*. Three examples are as follows:

```
#(1)
def function():                         #(a)
        print("Hello, world!")          #(b)

#(2)
if x:                                   #(a)
        print("Hello, world!"           #(b)

#(3)
while True:                             #(a)
        print("Hello, world!")          #(b)
```

If the exact meaning of these operations doesn't quite make sense to you, don't worry too much, that will be covered. The important thing to notice is that for any such operation, the inner code to be executed is *always* indented by one tab level more than the code that dictates the conditions of the code's execution (a level is the blank space created by a tab input). This isn't just to make your code look organised, it is a rule that the Python interpreter follows in order to know how to execute code. If you have ever seen another programming language like Java or C, it is analogous to enclosing the code to be executed in curly braces "" following its calling code. Code blocks can be "nested" inside of other code blocks using the same rule. An example is as follows:

```
def function(): #block 1
        y = 2 #block 1
        x = 0 #block 1
        if x = 0: #block 2
                y = 0 #block 2
                while y < 2: #block 3
                        print("Hello, world!") #block 3
                        y += 1 #block 3
                        day = "monday" #block 3
                print(day) #block 2
function()
```

Notice how the if statement is on the same layer as x = 0? Please be careful and don't confuse when I said "the inner code to be executed (b) is *always* indented by one tab level" to mean that "inner code is only something that comes after (a) the code that dictates the conditions of the code's execution". Absolutely not. In this case the if statement becomes "the inner code to be executed (b)" because we want it to run after assignment x = 0 has occurred in the scope of the function definition.

Scope

So far variables and variable assignment has been covered, but we haven't discussed *when* a variable can be written to or read from. Scope is the set of rules that dictate this behaviour. In short, a scope rule defines a sort of variable "enclosure" in a way meaning that variables cannot be accessed across the borders of these enclosures without special conditions allowing them to. Examples of such "enclosures" are separate functions, and inside and outside of functions. In fact these two scenarios deal with *local* and *global* scope. In Python there are in fact four scope types: local, enclosed, global and built-in. In this guide we shall only be dealing with local and global.

Local and global scope

As mentioned above, scopes define disconnected enclosures between which variables cannot be accessed in the ways we have seen so far. This means that there are special mechanisms in place so that this can be done

Scope nesting

In Python, a block does not define a scope, unlike in many other programming languages. This means that you can define a variable within a nested block and then call it outside of that block.

```
def function():
        y = 2
        x = 0
        if x = 0:
                y = 0
                while y < 2:
                        print("Hello, world!")
                        y += 1
                        day = "monday"
                print(day)
function()
```

If you were to run this script the output would look like this:

```
Hello, world!
Hello, world!
monday
```

As you can see the print function is called from a different block to where the variable it is printing is defined, but because it was defined before it was accessed by the print function Python considers it to be properly defined and available to be passed as an argument to a function

Scope reaching variable definitions

```
def function():
    y = 2
    x = 0
    if x == 1:
        y = 0
        while y < 2:
            print("Hello, world!")
            y += 1
            day = "monday"
    else:
        print(day)

function()
```

Scope order

Modules

Versions

Python is currently being maintained in two different versions: Python 2.7 and Python 3.7. Bear in mind that Python 2.7's lifecycle ends in 2020 and will not be maintained past then. I won't dwell too long on the differences as the important ones are well documented elsewhere, as such as **here** however I will mention a couple of key differences and important things that a beginning absolutely must know.

Syntax and operations

First of all in Python 2.x, 'print' is a statement, meaning that it is formatted like:

```
print 2
print "text"
print variable
```

Whereas in python 3.x 'print' is a function which means it takes an argument, and should look like this:

```
print(2)
print("text")
print(variable)
```

The other difference I will mention relates to the division operator, and whether it will return an integer or a float. The following operations in Python 2.x function like this:

```
3/2 = 1
3//2 = 1
3/2.0 = 1.5
3//2.0 = 1.0
```

Whereas in python 3.x the following operations function like this:

```
3/2 = 1.5
3//2 = 1
3/2.0 = 1.5
3//2.0 = 1.0
```

As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.As you can see the concept of integer divison with rounding must be explicitly defined in Python 3.x.

Modules and running scripts

As you may expect different versions of python may have compatibility issues with different modules, although there are some that have been developed with the intention to run on both. This compatibility may be dependent on that module's version as well so ensure that you check the developer's documentation and download the correct version. When it comes to running your script you will have to tell your IDE which version of the Python interpreter you want to use if you are using something like Pycharm. If you are using a text editor you must specify which version of Python you want to use as the interpreter for your script. An example is as follows:

```
python my_script.py
python3 my_script.py
```