

Wolf Pack course on Python

V 1.0

Introductory Python

By Arabindo (email: arabindo@pm.me)

Created for The Bosons (<https://bit.ly/bosonsD>)

Disclaimer: Here, I have written this report in a simple way even sometimes with some analogy, that help me to remember those concepts/ideas. It might go wrong in some places. Any sort of suggestions will be appreciated.

V1.0.1

Introduction, Variables and Operations

Dated: Feb 18, 2021 (Edit: Feb 21)

1. Introduction

From birth, the primary objective of every human is how they could reduce their effort. The idea of computer programming is just another milestone to reach the effort. It might be a little difficult at the beginning, but the time being it leads us to mastery of laziness. After all, it has all the power to repeat the same thing over and over again without being tired.

Another reason to learn programming might be to become “cool” :P

However, choosing Python as the first programming language has numerous advantages with “No disadvantages”, it illustrates almost every programming concept in a lucid manner. Although using python everywhere might be a headache, one must learn other languages too as per the requirements. But growing Python community and libraries make it a little simpler. Nowadays, *almost* everywhere, people do use python with some exception. But that, not the concern for now. What I understand, heading with python will give us the power to become lazier (as a function of time).

I found one particular book in this context, which is pretty nice: *Automate the Boring Stuff with Python, 2nd Edition: Practical Programming for Total Beginner* by Al Sweigart

There are different types of programming language, depending on the various parameter.

For example, if we want to classify the languages by the human readability, then we have only two types of language -

1. High-Level Language: This kind of languages are very intuitive and human-readable. The only disadvantage, there is some extra processing time to convert the language in machine-readable form, that is~
2. Low-Level Language: This kind of languages are difficult to understand for the human being. The advantage is, these languages interact with the machine(kernel, the core of an operating system) directly.

Depending on how the program gets translated into machine-readable form, further, we have two types of the program process

1. Compiler: Here, the whole program gets converted into machine-readable form, and the compiler throws all the *syntactical error* at once. For example, C/C++ this kind languages uses the compiler processing program (yes compiler and interpreter, is itself another program)
2. Interpreter: Here the program gets executed line by line. Therefore, if the interpreter encounters an error on some particular line of a program, then it will show the error and stop executing. As a result, the languages that rely on interpreter processing is a little slow compared to those, which language use compiler processing scheme. Python is a classic example, which uses the interpreter scheme.

Another way of classifying a language depends on the feature of the language. If one has to focus more on the way the program will be executed, then language is known as *Procedural Language*. On the contrary, if one has to focus the how the user data will be processed, the

language is known as Object-Oriented Language. C language is an example of procedural language, whereas c++, python are object-oriented language.

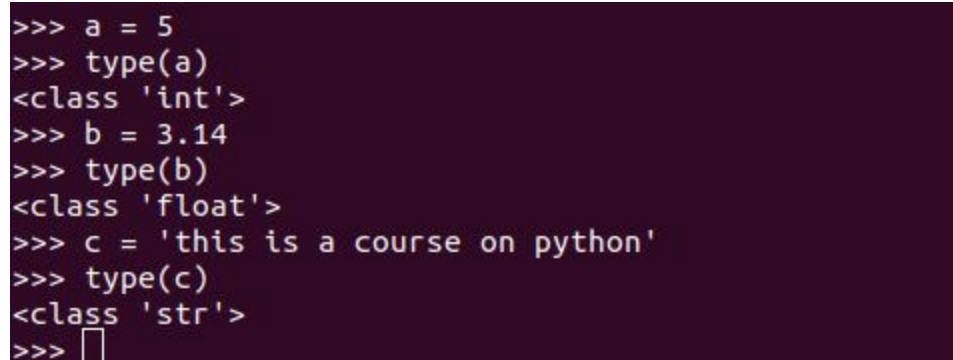
2. Variable

To do something repeatedly, we do need variables that can be updated at a later time. Suppose, I said, I have four storybook and someone, who is reading this, is generous enough to offer me another book (for no reason). So where I used to keep counting the numbers, I will update it to a new number five.

There can be different type of variables, to *remember* a different type of data. Like to remember five, we can use an **integer** type variable, to remember the name of the book, we can use a variable of type **string**, to remember the value of pi = 3.14159265, I will use a **float** type variable (Not to be confused with decimal - is not a concern at the moment, but an interested reader can consult <https://docs.python.org/3/library/decimal.html>).

The best part of python, we really do not need to mention, the type of the variable, she will guess it for us. Suppose, we define a variable x which can remember the name of this wolf pack course 'wolf-pack-python'. Then x by default will be interpreted as a string type variable.

Note: String type *object* are generally enumerated within the double or single inverted comma



```
>>> a = 5
>>> type(a)
<class 'int'>
>>> b = 3.14
>>> type(b)
<class 'float'>
>>> c = 'this is a course on python'
>>> type(c)
<class 'str'>
>>> 
```

Fig 1

3. Operations and Statements

The primary goal of the computer is to ease the computation(calculations), so the different type of operations (basically refer to arithmetic and logical) are fundamental.

Just as in mathematics, here we also have four arithmetic operations - addition, subtraction, multiplication and division. In addition, we have a *modulo* operator.



```
>>> 2+3
5
>>> 6*7
42
>>> 15/5
3.0
>>> 18%5
3
```

Fig 2

There are few more arithmetic operations and a class of logical operation. It'll be better if we postpone the discussion of those until we encounter them for a specific purpose. That'll definitely bring more clarity to those kinds of stuff.

If we want our program do some job, we must instruct the interpreter to do so. The set of instructions form a program. These instructions are often written in a statement format. Like if we want to print the value variable(say x) we will write *print(x)*. We can also print a value directly like, *print(92)* or maybe a string *print("Hey! How are you?")*. We can also do operations within the print statement, like *print(2+5)*. While we are assigning a value just like in figure 1, we are essentially stating the fact, *hey Python, remember that 'a' means 5* and so on.

```
>>> x = "hey dude"
>>> print(x)
hey dude
>>> print(92)
92
>>> print(2+5)
7
>>> a = 'addition of' + ' string' + ' is also possible' + ' in Python'
>>> print(a)
addition of string is also possible in Python
>>> █
```

Fig 3

We can also have statements like *input* to get user input and then we can assign it to some variable. For example-

```
>>> example = input("enter the name: ")
enter the name: Arabindo
>>> print(example)
Arabindo
>>> █
```

Fig 4

A thing worth to mention, by default, Python take user input as a string. So it really doesn't matter if we entering an integer, float or maybe string. To get rid of this, we will be using some ***built-in functions***.

Functions

Dated: Feb 22, 2021

4. General Overview on Functions

Last time we reviewed some basic concepts on programming, but the real fun begins with functions, conditions and iterative loops. This time we will be looking into functions.

There are basically two types of functions

1. Built-in Function: Every time we install a new interpreter/compiler program on our machine, it comes with various *functions* and *libraries*, that often characterize the core features of a programming language.
2. User-Defined Function: From the name, it is very intuitive. A function defined by the programmer(user).

But wait, what is a function at all?

Functions are like a machine (One who is acquainted with highschool mathematics, can think just like a mathematical function), that take input(s), by some means, and give the desired output. Functions just contain a set of statements that are designed to do some specific task. And we can use those functions anytime at any number of time within our program. Functions have various advantages, which'll be clear as we start using them.

4.1 Built-in Function

We have already used some of the built-in functions as the statement, like *print()*, *input()*, *type()*. We put the parameter that will be used by the functions as an input within the parentheses. For example, when we are stating that ***type(a)***, the variable **a** is passing in the function **type** as a parameter. The working principle will be more clear with~

4.2 User-Defined Function

Here, the programmer has to define a function manually. Let's start with an example.

Suppose we want to convert minutes into second. One obvious way to do:

```
>>> minute = 2
>>> second = minute * 60
>>> print(second)
120
>>>
```

Fig 5

But the problem with this little 'seem-to-easy' two-line code is that it can not be reused. So every time we need to convert the minute into the second, we need to write this script over and over again. Fortunately, we have a clever way to deal with this.

We can define a function named **converter**, in the following manner

```
>>> def converter(t):
...     c = t * 60
...     return c
...
>>> seconds = converter(10)
>>> print(seconds)
600
>>> print(converter(50))
3000
```

Fig 6

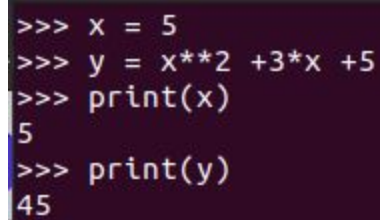
Here we have to define a function by the *keyword* **def**, and then one has to state the instructions in a new *code block*.

By Code Block, we mean a certain number of the statement must be evaluated by the computer under some other statement(s) - here **def**. All programming language have their own way to show a code block, in the case of python we intend the statement by 4 space(or 1 tab).

We pass the parameter **minute** to the function, that characterizes the function. Then we did all the necessary calculation and return them to the *main* program(function) by the statement **return**.

Here we go with another example. Suppose, we need to evaluate a mathematical function $x^2 + 3x + 5$ for different value of x within the programme multiple time.

One easy and obvious way to do this



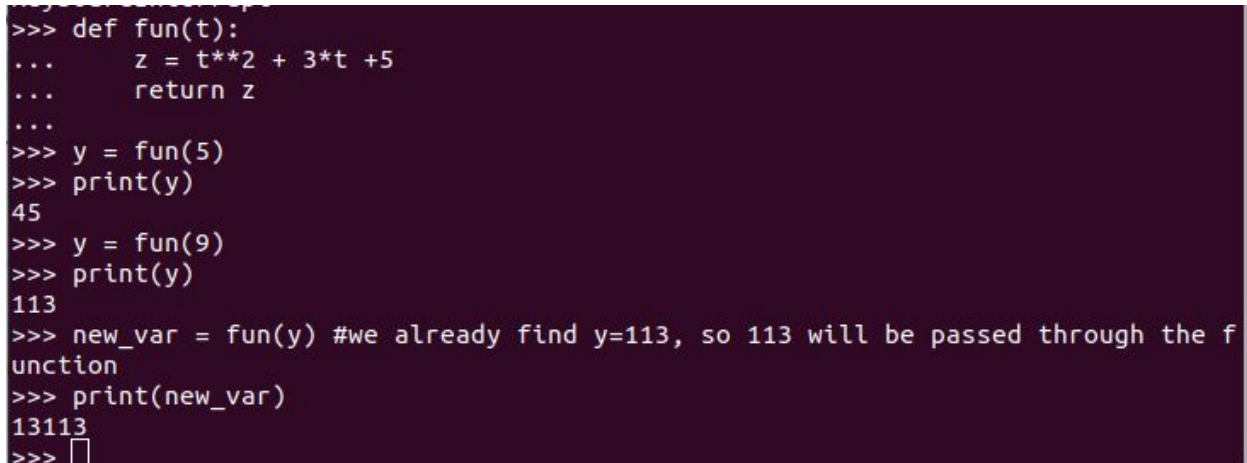
```
>>> x = 5
>>> y = x**2 + 3*x + 5
>>> print(x)
5
>>> print(y)
45
```

Fig 7

In this case, we have to write the expression over and over again, every time we need to find the value of the expression for some **x**.

Here we've introduced a new arithmetic operator ******, It is a raising operator. For example, if we write $5**6$ it'll be evaluated as 6 times 5 that is $5 \times 5 \times 5 \times 5 \times 5 \times 5 = 15625$

Fortunately, we have a cleaver way to do this same job.



```
>>> def fun(t):
...     z = t**2 + 3*t + 5
...     return z
...
>>> y = fun(5)
>>> print(y)
45
>>> y = fun(9)
>>> print(y)
113
>>> new_var = fun(y) #we already find y=113, so 113 will be passed through the f
unction
>>> print(new_var)
13113
>>> 
```

Fig 8

5. Nature of Variable

In general, there are two scopes of a variable

1. *Local Scope*: Whenever we define a variable, the scope of a variable is limited to the function where we are defining the variable. If we define a variable outside of a function, that actually corresponds to the main function only. In the below example, one must appreciate the fact, the minute that declared outside of the *converter* function is different from the minute in the main function. The interpreter will *point* to a different location on the memory (hardware) for these two different minute variable.

```
>>> def converter(minute):
...     sec = minute*60
...     return sec
...
>>> minute = 10
>>> print(converter(minute))
600
>>>
```

Fig 9

The standard practice among the programmer is to use a different name for variables, so later it must not confuse the *programmer*, just like as we did in figure 6.

2. *Global Scope*: Sometimes it is convenient to define a variable in such a way, that can be used from any other function. It can be easily done just with a keyword **global**. The below example illustrates the idea.

```
>>> global x
>>> x = 10
>>> def func(new):
...     global x #here we again declare the scope, otherwise interpreter will think x belongs to local scope
...     x = new + x
...     return 0
...
>>> print(x)
10
>>> func(5)
0
>>> print(x)
15
>>>
```

Fig 10

In the above figure, the first print line shows 10, that is **x** is unchanged since the **func** haven't executed yet. In the next line, we call the *func*, then it got executed, since the function returns 0, it shows 0 in the output line. But, the **x** has been changed. So the next time when we print x, it shows the updated value.

Just give it a thought, if we don't have the function *print()*, we had to write the whole program for printing a simple number or a string. Then the whole program would be a mess. This *simple* function *print()* shows the necessity of a function, whether it's in-built or user-defined.

=====will continue=====