You have **2** free member-only stories left this month. Sign up for Medium and get an extra one

Everything About Python — Beginner To Advanced

Everything You Need To Know In One Article

Farhad Malik Follow
Mar 9, 2019 · 30 min read ★

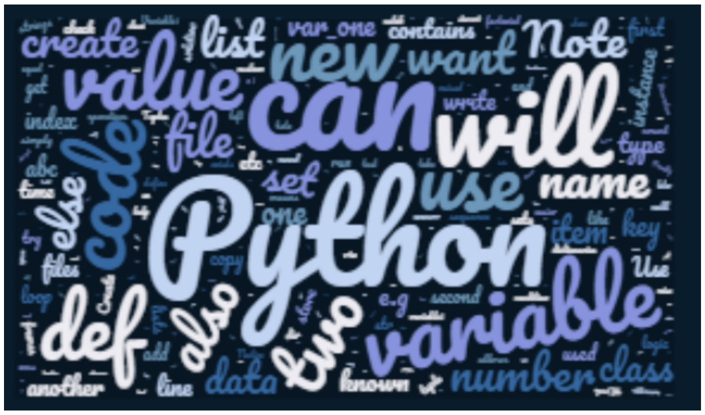
This article aims to outline all of the key points of the Python programming language. My target is to keep the information **short, relevant, and focus** on the most important topics which are absolutely required to be understood.

After reading this blog, you will be able to use any Python library or implement your own Python packages.

You are not expected to have any prior programming knowledge and it will be very quick to grasp all of the required concepts.

I will also highlight top discussion questions that people usually query regarding Python programming language.

> Lets build the knowledge gradually and by the end of the article, you will have a thorough understanding of Python.



> This article contains 25 key topics. Let's Start.

1. Introducing Python

What Is Python?

- Interpreted **high-level object-oriented dynamically-typed scripting** language.
- As a result, **run time errors** are usually encountered.

Why Python?

- Python is the most popular language due to the fact that it's easier to code and understand it.
- Python is an object-oriented programming language and can be used to write functional code too.
- It is a suitable language that bridges the gaps between business and developers.
- Subsequently, it takes less time to bring a Python program to market compared to other languages such as C#/Java.
- Additionally, there are a large number of python machine learning and analytical packages.
- A large number of communities and books are available to support Python developers.
- Nearly all types of applications, ranging from forecasting analytical to UI, can be implemented in Python.
- There is no need to declare variable types. Thus it is quicker to implement a Python application.

Why Not Python?

- Python is slower than C++, C#, Java. This is due to the lack of Just In Time optimisers in Python.
- Python syntactical white-space constraint makes it slightly difficult to implement for new coders.
- Python does not offer advanced statistical features as R does.
- Python is not suitable for low-level systems and hardware interaction.

How Does Python Work?

This image illustrates how python runs on our machines:

The key here is the Interpreter that is responsible for translating high-level Python language to low-level machine language.

The way Python works is as follows:

1. A Python virtual machine is created where the packages (libraries) are installed. Think of a virtual machine as a container.

2. The python code is then written in .py files
3. CPython compiles the Python code to bytecode. This bytecode is for the Python virtual machine.

> Now, this virtual machine is machine-dependent but the Python code isn't.

4. When you want to execute the bytecode then the code will be interpreted at runtime. The code will then be translated from the bytecode into the machine code. The bytecode is not dependent on the machine on which you are running the code. This makes Python machine-independent.

> Python byte code is Python virtual machine-dependent and this makes Python machine-independent.

The point to note is that we can write Python code in one OS, copy it to another OS and simply run it.

2. Variables — Object Types And Scope

- Variables store information that can be used and/or changed in your program. This information can be an integer, text, collection, etc.
- Variables are used to hold user inputs, local states of your program, etc.
- Variables have a **name** so that they can be referenced in the code.
- The fundamental concept to understand is that everything is an object in Python.

Python supports numbers, strings, sets, lists, tuples, and dictionaries. These are the standard data types. I will explain each of them in detail.

Declare And Assign Value To Variable

Assignment sets a value to a variable.

To assign variable a value, use the equals sign (=)

```
myFirstVariable = 1
mySecondVariable = 2
myFirstVariable = "Hello You"
```

- Assigning a value is known as **binding** in Python. In the example above, we have assigned the value of 1 to myFirstVariable.

Note how I assigned an integer value of 1 and then a string value of "Hello You" to the same myFirstVariable variable. ***This is possible due to the fact that the data types are dynamically typed in python.***

This is why Python is known as a dynamically typed programming language.

If you want to assign the same value to more than one variables then you can use the chained assignment:

```
myFirstVariable = mySecondVariable = 1
```

Numeric

- Integers, decimals, floats are supported.

```
value = 1 #integer
value = 1.2 #float with a floating point
```

- Longs are also supported. They have a suffix of L e.g. 9999999999999L

Strings

- Textual information. Strings are sequence of letters.
- A string is an array of characters
- A string value is enclosed in quotation marks: single, double or triple quotes.

```
name = 'farhad'
name = "farhad"
name = """farhad"""
```

- Strings are immutable. Once they are created, they cannot be changed e.g.

```
a = 'me'

Updating it will fail:
a[1]='y'

It will throw a Type Error
```

- When string variables are assigned a new value then internally, Python creates a new object to store the value.

Therefore a reference/pointer to an object is created. This pointer is then assigned to the variable and as a result, the variable can be used.

We can also assign one variable to another variable. All it does is that a new pointer is created which points to the same object:

```
a = 1 #new object is created and 1 is stored there, new pointer is
created, the pointer connects a to 1
b = a #new object is not created, new pointer is created that
connects b to 1
```

Variables can have local or global scope.

Local Scope

- Variables declared within a function, as an example, can only exist within the block.
- Once the block exists, the variables also become inaccessible.

```
def some_funcion():
  TestMode = False

print(TestMode) <- Breaks as the variable doesn't exist outside
```

In Python, if-else and for/while loop block doesn't create any local scope.

```
for i in range(1, 11):
    test_scope = "variable inside for loop"

print(test_scope)
```

Output:

```
variable inside for loop
```

With if-else block

```
is_python_awesome = True

if is_python_awesome:
    test_scope = "Python is awesome"

print(test_scope)
```

Output:

```
Python is awesome
```

Global Scope

- Variables that can be accessed from any function have a global scope. They exist in the __main__ frame.
- You can declare a global variable outside of functions. It's important to note that to assign a global variable a new value, you will have to use the "*global*" keyword:

```
TestMode = True
def some_function():
  global TestMode
  TestMode = False

some_function()
print(TestMode) <--Returns False
```

Removing the line "global TestMode" will only set the variable to False within the some_function() function.

Note: Although I will write more on the concept of modules later, but if you want to share a global variable across a number of modules then you can create a shared module file e.g. configuration.py and locate your variable there. Finally, import the shared module in your consumer modules.

Finding Variable Type

- If you want to find the type of a variable, you can implement:

```
type('farhad')
--> Returns <type 'str'>
```

Comma In Integer Variables

- Commas are treated as a sequence of variables e.g.

```
9,8,7 are three numeric variables
```

3. Operations

- Allows us to perform computation on variables

Numeric Operations

- Python supports basic `*`, `/`, `+`, `-`
- Python also supports floor division

```
1//3  #returns 0
1/3 #returns 0.333
```

- Note: the return type of division is always **float** as shown below:

```
a = 10/5
print(type(a)) #prints float
```

- Additionally, python supports exponentiation via ** operator:

```
2**3 = 2 * 2 * 2 = 8
```

- Python supports Modulus (remainder) operator too:

```
7%2 = 1
```

There is also a divmod in-built method. It returns the divider and modulus:

```
print(divmod(10,3)) #it will print 3 and 1 as 3*3 = 9 +1 = 10
```

String Operations

Concat Strings:

```
'A' + 'B' = 'AB'
```

Remember string is an immutable data type, therefore, concatenating strings creates a new string object.

Repeat String:

```
'A'*3 will repeat A three times:  AAA
```

Slicing:

```
y = 'Abc'
y[:2] = ab
y[1:] = bc
y[:-2] = a
y[-2:] = bc
```

Reversing:

```
x = 'abc'
x = x[::-1]
```

Negative Index:

If you want to start from the last character then use negative index.

```
y = 'abc'
print(y[:-1]) # will return ab
```

It is also used to remove any new line carriages/spaces.

Each element in an array gets two indexes:

- From left to right, the index starts at 0 and increments by 1
- From right to left, the index starts at -1 and decrements by 1
- Therefore, if we do y[0] and y[-len(y)] then both will return the same value: ‘a’

```
y = 'abc'
print(y[0])
print(y[-len(y)])
```

Finding Index

```
name = 'farhad'
index = name.find('r')

#returns 2
```

```
name = 'farhad'
index = name.find('a', 2) # finds index of second a

#returns 4
```

For Regex, use:

- split(): splits a string into a list via regex
- sub(): replaces matched string via regex
- subn(): replaces matched string via regex and returns number of replacements

Casting

- str(x): To string
- int(x): To integer
- float(x): To floats

Set Operations

- A set is an unordered data collection without any duplicates. We can define a set variable as:

```
set = {9,1,-1,5,2,8,3, 8}
print(set)
```

This will print: {1, 2, 3, 5, 8, 9, -1}

Note duplicates are removed.

Set has *a item in set, len(set)* and *for item in set* operations. However it does not support indexing, slicing like lists.

Some of the most important set operations are:

- set.add(item) — adds item to the set
- set.remove(item) — removes item from the set and raises error if it is not present
- set.discard(item) — removes item from the set if it is present
- set.pop() — returns any item from the set, raises KeyError if the set is empty
- set.clear() clears the set

Intersect Sets

- To get what's common in two sets

```
a = {1,2,3}
b = {3,4,5}
c = a.intersection(b)
```

Difference In Sets

- To retrieve the difference between two sets:

```
a = {1,2,3}
b = {3,4,5}
c = a.difference(b)
```

Union Of Collections

- To get a distinct combined set of two sets

```
a = {1,2,3}
b = {3,4,5}
c = a.union(b)
```

Ternary Operator

- Used to write conditional statements in a single line.

Syntax:

[If True] if [Expression] Else [If False]

For example:

```
Received = True if x == 'Yes' else False
```

4. Comments

Single Line Comments

```
#this is a single line comment
```

Multiple Line Comments

One can use:

~~~
```this is a multi
line
comment```
~~~

5. Expressions

Expressions can perform boolean operations such as:

- Equality: ==
- Not Equal: !=
- Greater: >
- Less: <
- Greater Or Equal >=
- Less Or Equal <=

6. Pickling

Converting an object into a string and dumping the string into a binary file is known as pickling. The reverse is known as unpickling.

7. Functions

- Functions are sequence of statements that you can execute in your code. If you see repetition in your code then create a reusable function and use it in your program.
- Functions can also reference other functions.
- Functions eliminate repetition in your code. They make it easier to debug and find issues.
- Finally, functions enable code to be understandable and easier to manage.
- In short, functions allow us to split a large application into smaller chunks.

Define New Function

```
def my_new_function():
  print('this is my new function')
```

Calling Function

```
my_new_function()
```

Finding Length Of String

Call the len(x) function

```
len('hello')
prints 5
```

Arguments

- Arguments can be added to a function to make the functions generic.
- This exercise is known as generalization.
- You can pass in variables to a method:

```
def my_new_function(my_value):
  print('this is my new function with ' + my_value)
```

- **Optional arguments:**

We can pass in optional arguments by providing a default value to an argument:

```
def my_new_function(my_value='hello'):
  print(my_value)

#Calling
my_new_function() => prints hello
my_new_function('test') => prints test
```

- ***arguments:**

If your function can take in any number of arguments then add a * in front of the parameter name:

```
def myfunc(*arguments):
  return a


myfunc(a)
myfunc(a,b)
myfunc(a,b,c)
```

- ****arguments:**

It allows you to pass a varying number of keyword arguments to a function.

You can also pass in dictionary values as keyword arguments.

```
def test(*args, **kargs):
    print(args)
    print(kargs)
    print(args[0])
print(kargs.get('a'))



alpha = 'alpha'
beta = 'beta'
test(alpha, beta, a=1, b=2)
```

This will print:

(3, 1)
('alpha', 'beta')
{'a': 1, 'b': 2}
alpha
1

Return

- Functions can return values such as:

```
def my_function(input):
  return input + 2
```

- If a function is required to return multiple values then it's suitable to return a tuple (comma separated values). I will explain tuples later on:

```
resultA, resultB = get_result()

get_result() can return ('a', 1) which is a tuple
```

Lambda

- Single expression anonymous function.
- It is an inline function.

```
my_lambda = lambda x,y,z : x - 100 + y - z

my_lambda(100, 100, 100) # returns 0
```

Syntax:

variable = lambda arguments: expression

Lambda functions can be passed as arguments to other functions.

Object Identity

I will attempt to explain the important subject of Object Identity now.

Whenever we create an object in Python such as a variable, function, etc, the underlying Python interpreter creates a number that uniquely identifies that object. Some of the objects are created up-front.

When an object is not referenced anymore in the code then it is removed and its identity number can be used by other variables.

dir() and help()

- dir() -displays defined symbols
- help() — displays documentation

Let's understand it in detail:

Consider the code below:

```
var_one = 123
def func_one(var_one):
    var_one = 234
    var_three = 'abc'


var_two = 456
print(dir())
```

var_one and var_two are the two variables that are defined in the code above. Along with the variables, a function named func_one is also defined. An important note to keep in mind is that everything is an object in Python, including a function.

Within the function, we have assigned the value of 234 to var_one and created a new variable named var_three and assigned it a value of 'abc'.

Now, let's understand the code with the help of dir() and id()

The above code and its variables and functions will be loaded in the Global frame. The global frame will hold all of the objects that the other frames require. As an example, there are many built-in methods loaded in Python that are available to all of the frames. These are the function frames.

Running the above code will print:

```
['__annotations__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'func_one', 'var_one', 'var_two']
```

The variables that are prefixed with __ are known as the special variables.

Notice that the var_three is not available yet. Let's execute func_one(var_one) and then assess the dir():

```
var_one = 123
def func_one(var_one):
    var_one = 234
    var_three = 'abc'

var_two = 456
func_one(var_one)
print(dir())
```

We will again see the same list:

```
['__annotations__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'func_one', 'var_one', 'var_two']
```

This means that the variables within the func_one are only within the func_one. When func_one is executed then a Frame is created. Python is top-down therefore it always executes the lines from top to the bottom.

The function frame can reference the variables in the global frame but any other function frame cannot reference the same variables that are created within itself. As an instance, if I create a new function func_two that tries to print var_three then it will fail:

```
var_one = 123
def func_one(var_one):
    var_one = 234
    var_three = 'abc'

var_two = 456


def func_two():
    print(var_three)

func_one(var_one)
func_two()
print(dir())
```

We get an error that **NameError: name 'var_three' is not defined**

What if we create a new variable inside func_two() and then print the dir()?

```
var_one = 123
def func_one(var_one):
    var_one = 234
    var_three = 'abc'

var_two = 456


def func_two():
    var_four = 123
    print(dir())
```

```
func_two()
```

This will print var_four as it is local to func_two.

How does Assignment work In Python?

This is by far one of the most important concepts to understand in Python. Python has an id() function.

When an object (function, variable, etc.) is created, CPython allocates it an address in memory. The id() function returns the “identity” of an object. It is essentially a unique integer.

As an instance, let’s create four variables and assign them values:

```
variable1 = 1
variable2 = "abc"
variable3 = (1,2)
variable4 = ['a',1]

#Print their Ids
print('Variable1: ', id(variable1))
print('Variable2: ', id(variable2))
print('Variable3: ', id(variable3))
print('Variable4: ', id(variable4))
```

The ids will be printed as follows:

Variable1: 1747938368
Variable2: 152386423976
Variable3: 152382712136
Variable4: 152382633160

Each variable has been assigned a new integer value.

The first assumption is that whenever we use the assignment “=” then Python creates a new memory address to store the variable. Is it 100% true, not really!

I am going to create two variables and assign them to the existing variables.

```
variable5 = variable1
variable6 = variable4

print('Variable1: ', id(variable1))
print('Variable4: ', id(variable4))
print('Variable5: ', id(variable5))
print('Variable6: ', id(variable6))
```

Python printed:

Variable1: **1747938368**
Variable4: 819035469000
Variable5: **1747938368**
Variable6: 819035469000

Notice that Python did not create new memory addresses for the two variables. This time, it pointed both of the variables to the same memory location.

Let’s set a new value to the variable1. Remember 2 is an integer and integer is an immutable data type.

```
print('Variable1: ', id(variable1))
variable1 = 2
print('Variable1: ', id(variable1))
```

This will print:

Variable1: 1747938368
Variable1: 1747938400

It means whenever we use the = and assign a new value to a variable that is not a variable then internally a new memory address is created to store the variable. Let’s see if it holds!

What happens when the value is a mutable data type?

variable6 is a list. Let’s append an item to it and print its memory address:

```
print('Variable6: ', id(variable6))
variable6.append('new')
print('Variable6: ', id(variable6))
```

Note that the memory address remained the same for the variable as it is a mutable data type and we simply updated its elements.

Variable6: 678181106888
Variable6: 678181106888

Let’s create a function and pass a variable to it. If we set the value of the variable inside the function, what will it do internally? let’s assess

```
def update_variable(variable_to_update):
    print(id(variable_to_update))
update_variable(variable6)
print('Variable6: ', id(variable6))
```

We get:

678181106888
Variable6: 678181106888

Notice that the id of variable_to_update points to the id of the variable 6.

This means that if we update the variable_to_update in a function and if variable_to_update is a mutable data type then we’ll update variable6.

```
variable6 = ['new']
print('Variable6: ', variable6)

def update_variable(variable_to_update):
    variable_to_update.append('inside')


update_variable(variable6)
print('Variable6: ', variable6)
```

This printed:

Variable6: [‘new’]
Variable6: [‘new’, ‘inside’]

It shows us that the same object is updated within the function as it was expected by both having the same ID.

If we assign a new value to a variable, regardless of if it’s immutable and mutable data type then the change will be lost once we come out of the function:

```
print('Variable6: ', variable6)

def update_variable(variable_to_update):
    print(id(variable_to_update))
    variable_to_update = ['inside']


update_variable(variable6)
print('Variable6: ', variable6)
```

Variable6: [‘new’]
344115201992
Variable6: [‘new’]

Now an interesting scenario: Python doesn’t always create a new memory address for all new variables.

Finally, what if we assign two different variables a string value such as ‘a’. Will it create two memory addresses?

```
variable_nine = "a"
variable_ten = "a"
print('Variable9: ', id(variable_nine))
print('Variable10: ', id(variable_ten))
```

Notice, both the variables have the same memory location:

Variable9: 792473698064
Variable10: 792473698064

What if we create two different variables and assign them a long string value:

```
variable_nine = "a"*21
variable_ten = "a"*21
print('Variable9: ', id(variable_nine))
print('Variable10: ', id(variable_ten))
```

This time Python created two memory locations for the two variables:

Variable9: 541949933872
Variable10: 541949933944

This is because Python creates an internal cache of values when it starts up. This is done to provide faster results. It creates a handful of memory addresses for small integers such as between -5 to 256 and smaller string values. This is the reason why both of the variables in our example had the same ID.

Photo by Noah Silliman on Unsplash

== vs is

Sometimes we want to check whether two objects are equal.

- If we use == then it will check whether the two arguments contain the same data
- If we use *is* then Python will check whether the two objects refer to the same object. The id() needs to be the same for both of the objects

```
var1 = "a"*30
var2 = "a"*30
print('var1: ', id(var1)) #318966315648
print('var2: ', id(var2)) #168966317364

print('== :', var1 == var2) #returns True
print('is :', var1 is var2) #returns False
```

8. Modules

What is a module?

- Python is shipped with over 200 standard modules.
- A module is a component that groups similar functionality of your python solution.
- Any python code file can be packaged as a module and then it can be imported.
- Modules encourage componentised design in your solution.
- They provide the concept of namespaces to help you share data and services.
- Modules encourage code reusability and reduce variable name clashes.

PYTHONPATH

- This environment variable indicates where the Python interpreter needs to navigate to locate the modules. PYTHONHOME is an alternative module search path.

How To Import Modules?

- If you have a file: MyFirstPythonFile.py and it contains multiple functions, variables and objects then you can import the functionality into another class by simply doing:

```
import MyFirstPythonFile
```

- Internally, python runtime will compile the module's file to bytes and then it will run the module code.
- If you want to import everything in a module, you can do:

```
import my_module
```

- If your module contains a function or object named my_object then you will have to do:

```
print(my_module.my_object)
```

Note: If you do not want the interpreter to execute the module when it is loaded then you can check whether the __name__ == '__main__'

2. From

- If you only want to access an object or parts of a module from a module then you can implement:

```
from my_module import my_object
```

- This will enable you to access your object without referencing your module:

```
print(my_object)
```

- We can also do from * to import all objects

```
from my_module import *
```

Note: Modules are only imported on the first import.

- **If you want to use a Python module in C:**

Use PyImport_ImportModule(module_name)

Namespace — two modules with same object name:

Use import over from if we want to use the same name defined in two different modules.

9. Packages

- Package is a directory of modules.
- If your Python solution offers a large set of functionalities that are grouped into module files then you can create a package out of your modules to better distribute and manage your modules.
- Packages enable us to organise our modules better which helps us in resolving issues and finding modules easier.
- Third-party packages can be imported into your code such as pandas/sci-kit learn and tensor flow to name a few.
- A package can contain a large number of modules.
- If our solution offers similar functionality then we can group the modules into a package:

```
from packageroot.packagefolder.mod import my_object
```

- In the example above, packageroot is the root folder. packagefolder is the subfolder under packageroot. my_module is a module python file in the packagefolder folder.
- Additionally, the name of the folder can serve as the namespace e.g.

```
from data_service.database_data_service.microsoft_sql.mod
```

Note: Ensure each directory within your package import contains a file __init__.py.

Feel free to leave the files blank. As __init__.py files are imported before the modules are imported, you can add custom logic such as start service status checks or to open database connections, etc.

PIP

- PIP is a Python package manager.
- Use PIP to download packages:

```
pip install package_name
```

10. Conditions

- To write if then else:

```
if a == b:
  print 'a is b'
elif a < b:
  print 'a is less than b'
elif a > b:
  print 'a  is greater than b'
else:
  print 'a is different'
```

*Note how **colons and indentations** are used to express the conditional logic.*

Checking Types

```
if not isinstance(input, int):
  print 'Expected int'
  return None
```

You can also add conditional logic in the else part. This is known as nested condition.

```
#let's write conditions within else
else:
 if a == 2:
    print 'within if of else'
 else:
     print 'within else of else'
```

11. Loops

While

- Provide a condition and run the loop until the condition is met:

```
while (input < 0):
 do_something(input)
 input = input-1
```

For

- Loop for a number of times

```
for  i in range(0,10)
```

- Loop over items or characters of a string

```
for letter in 'hello'
  print letter
```

One-Liner For

Syntax:

```
[Variable] AggregateFunction([Value] for [item] in [collection])
```

Yielding