# **Python Functions and Methods**

Comparison **Between Methods** and **Functions in Python.**

**Python method** is called on an object, unlike a **function**. Since we call a **method** on an object, it can access the data within it. A **method** may alter an object's state, but **Python function** usually only operates on it, and then prints something or returns a value.

## **Function**

A function is a block of code to carry out a specific task, will contain its own scope and is called by name. All functions may contain zero(no) arguments or more than one arguments. On exit, a function can or can not return one or more values.

## **Method**

A method in python is somewhat similar to a function, except it is associated with object/classes. Methods in python are very similar to functions except for two major differences.

* The method is implicitly used for an object for which it is called.
* The method is accessible to data that is contained within the class.

Function in Python can be thought of as a building block while writing a program, as the program builds up, functions help make the code organized and more readable and manageable. Functions also allow us to give a name to a block of code, hence allow us to run the block of code using the name assigned to the code anywhere it is called in the program, any number of times. We refer to that as calling a function.

Remember, functions only run when they are called to do so. We can pass data to a function, the data is known as arguments or parameters. A function can return data as a result or it can return nothing.

For instance, if we want to calculate a length of a list in Python, we can use a built-in len() function. Using any function simply means we are calling it to perform a task for which it is designed for.

**Types of Functions in Python**

i) Built-in functions

ii) User defined functions

iii) Anonymous functions (lambda functions)

* A built-in function such as print(), or input() on the standard input/output device. You can run a type() to check a data type of an object. Built-in functions are the functions that Python provides to accomplish common tasks.
* UserDefined functions: As the name suggests are custom functions to help resolve, achieve a particular task. This function is more customized to achieve an end goal.
* Anonymous functions on the other hand are also popularly known as lambda functions and are custom made without any name identifier. Lambda function can take any number of arguments, but can only have one expression.

**Revisiting built-in functions**

Built-in functions are the ones provided by Python. The very first built-in

function we had learned was the print() function to print a string given to it

as an argument on the standard output device. They can be directly used

within our code without importing any module and are always available

to use.

In addition to print function, We have learned the following built-in functions

until now in the previous sections.

• type(object) is used to check the data type of an object.

• input() is used to accept user input. It reads a line entered on a console by an input device such as a keyboard and converts it into a string and returned it

• float([value]) returns a floating point number constructed from a

number or string value.

• int([value]) returns an integer object constructed from a float or

string value, or return 0 if no arguments are given.

• round(number[, ndigits]) is used to round a float number up to digits

specified by ndigits.

• abs(value) returns the absolute value of a value provided as an argument.

• format(value[, format\_spec]) converts a value to a ’formatted’ representation,

as controlled by format\_spec.

• str([object]) returns a string version of object. If the object is not

provided, returns the empty string.

The append() method appends an element to the end of the list.

• bool([value]) return a Boolean value, i.e. one of True or False. value

is converted using the standard truth testing procedure1. If the value

is false or omitted, this returns False; otherwise, it returns True.

• dir([object]) returns the list of names in the current local scope

when an argument is not provided. With an argument, it attempts

to return a list of valid attributes for that object.

• len(object) returns the length (the number of items) of an object. The

argument may be a sequence (such as a string, bytes, tuple, list, or

range) or a collection (such as a dictionary, set, or frozen set).

It is worth noting that almost all built-in functions take one or more arguments,

perform the specific operation on it and return the output. We will

keep learning about many more built-in functions as we progress through

our Python learning journey. More information about various built-in functions

can be obtained from Python official documentation(https://docs.python.org/3/library/functions.html

).

In Python, a function is a group of related statements that performs a specific task.

Functions help break our programs into smaller and manageable modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

**A typical Syntax of a function in Python**

**def function\_name(parameters): # x, y, z # Parameters**

**“ “ “docstring“ “ “**

**more statement(s)**

**function\_name(1,2,3)# Arguments**

Above shown is a function definition that consists of the following components.

* Keyword **def** that marks the start of the function header.

A **function name** to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.

* Parameters (arguments) through which we pass values to a function. They are optional.
* A colon (**:**) to mark the end of the function header.
* Optional documentation string (docstring) to describe what the function does.
* One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
* An optional return statement to return a value from the function.

**A typical example of a function in Python**

**def greetings(name):**

**"""**

**This function greetings to**

**the person passed in as**

**a parameter**

**"""**

**print("Habari, " + name + ". Ya hasubuhi!")**

### **How to call a function in python?**

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

**>>> greetings(‘Bridgit’)**

**Habari, Bridgit, ya hasubuhi!**

**Note:** Now try running the above code in the Python program with the function definition to see the output.

def greetings(name):

"""

This function greets to

the person passed in as

a parameter

"""

print("Habari, " + name + ". Ya hasubuhi!")

greet('Jacky')

## **Docstrings**

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does. Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as the \_\_doc\_\_ attribute of the function.

**For example:**

Try running the following in the Python shell to see the output.

>>> print(greetings.\_\_doc\_\_)

This function greetings to

the person passed in as a parameter

**The return statement**

The return statement is used to exit a function and go back to the place from where it was called.

**A syntax of a return statement**

Return [expression\_list]

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

**For example:**

>>> print(greetings(“Bridgit”))

Habari, Jacky. Ya Asubuhi!

None

Here, None is the returned value since greetings() directly prints the name and no return statement is used.

### **Example of return**

def absolute\_value(number):

"""This function returns the absolute

value of the entered number"""

if number >= 0:

return number

else:

return -number

print(absolute\_value(4))

print(absolute\_value(-6))

**Output of the above program:**

**4**

**6**

### Example

def my\_function(x):

return 5 \* x

print(my\_function(3))

print(my\_function(5))

print(my\_function(9))

## **Arguments**

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (firstname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

### **Example**

# The wonderful Jackson 5

def my\_function(firstname):

print(firstname + " Jackson")

my\_function("Jackie")

my\_function("Tito")

my\_function("Jermaine")

my\_function("Marlon")

my\_function("Michael")

## **Parameters or Arguments?**

The terms *parameter* and *argument* can be used for the same thing: information that is passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

## **Number of Arguments**

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

### **Example**

This function expects 2 arguments, and gets 2 arguments:

**def my\_function(fname, lname):**

**print(fname + " " + lname)**

**my\_function("Beatrice", "Atieno")**

**Note:** If you try to call the function with 1 or 3 arguments, you will get an error:

**See an example below of the same.**

### **Example**

This function expects 2 arguments, but gets only 1:

def my\_function(fname, lname):

print(fname + " " + lname)

my\_function("Malcom")

## Arbitrary Arguments, \*args

If you are not sure of how many arguments that will be passed into your function declaration, simply add a **\*** before the parameter name in the function definition.

This way the function will receive a *tuple* of arguments, and can access the items accordingly:

### **Example**

If the number of arguments is unknown, add a **\*** before the parameter name:

def my\_function(**\***presidents):

print("The name of the first president of Ghana was: " + presidents[2])

my\_function("Dr. Kenneth Kaunda", "Mwalimu Julius Nyerere", "Dr. Kwame Nkrumah")

**Note:** *Arbitrary Arguments* are often shortened to *\*args* in Python documentations.

## 

## **Keyword Arguments**

We can also send arguments with the *key* = *value* syntax.

This way the order of the arguments does not matter.

**def my\_function(Engi3, Engi2, Engi1):**

**print("The youngest Scientist was: " + Engi3)**

**my\_function(Engi1 = "Galileo", Engi2 = "Newton", Engi3 = "Einstein")**

**Note:** The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

## **Arbitrary Keyword Arguments, \*\*kwargs**

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: **\*\*** before the parameter name in the function definition.

This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

### **Example**

If the number of keyword arguments is unknown, add a double **\*\*** before the parameter name:

**def my\_function(\*\*activist):**

**print("His last name was: " + activist["lname"])**

**my\_function(fname = "Malcolm", lname = "X")**

**Note:** *Arbitrary Kword Arguments* are often shortened to *\*\*kwargs* in Python documentations.

## 

## **Default Parameter Value**

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

### Example

**def my\_function(country = "Tanzania"):**

**print("I am from " + country)**

**my\_function("Kenya")**

**my\_function("Uganda")**

**my\_function()**

**my\_function("Somalia")**

## **The pass Statement**

function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

**def myfunction():**

**pass**

# having an empty function definition like this, would raise an error without # the pass statement

## **Passing a List as an Argument**

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

e.g. if you send a List as an argument, it will still be a List when it reaches the function:

### 

### 

### 

### **Example**

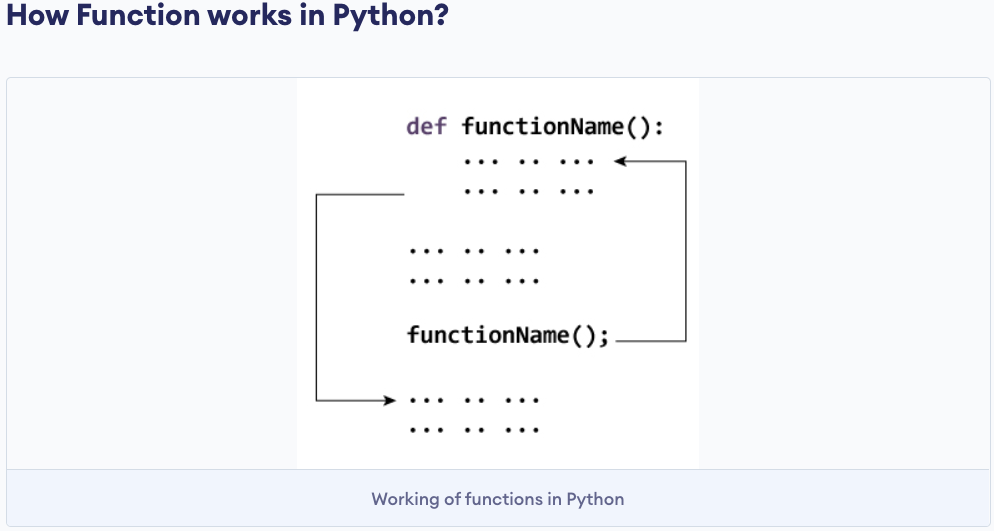
**def my\_function(politicians):**

**for x in politicians:**

**print(x)**

**mashuja = ["Tom Mboya", "J.M. Kariuki", "Argwings Kodhek","Dr.R. Ouko"]**

**my\_function(mashuja)**

****

## **Scope and Lifetime of variables**

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here below is an example to illustrate the scope of a variable inside a function.

def my\_func():

x = 10

print("Value inside function:",x)

x = 20

my\_func()

print("Value outside function:",x)

**Output of the above program:**

Value inside function: 10

Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my\_func() changed the value of x to 10, it did not affect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global**.

# **Python Anonymous/Lambda Function**

#### In this section, you'll learn about the anonymous functions, also known as lambda functions. You'll learn what they are, their syntax and how to use them (with examples).

## **What are lambda functions in Python?**

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the **def** keyword in Python, anonymous functions are defined using the **lambda** keyword.

Hence, anonymous functions are also called **lambda functions**.

### **Syntax of Lambda Function in python**

**lambda arguments: expression**

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

### **Example of Lambda Function in python**

Here is an example of lambda function that doubles the input value.

# Program to show the use of lambda functions

**double** = lambda x: x \* 2

print(double(5))

Program Output of the above

**10**

In the above program, lambda x: x \* 2 is the lambda function. Here **x** is the argument and **x \* 2** is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier double. We can now call it a normal function. The statement

**double** = lambda x: x \* 2

is nearly the same as:

def double(x):

return x \* 2

## **Use of Lambda Function in python**

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like **filter()**, **map()** etc.

### **Example use with filter()**

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of the filter() function to filter out only even numbers from a list.

**# Program to filter out only the even items from a list**

**my\_list = [1, 5, 4, 6, 8, 11, 3, 12]**

**new\_list = list(filter(lambda x: (x%2 == 0) , my\_list))**

**print(new\_list)**

Above Program Output below

[4, 6, 8, 12]

### **Example use with map()**

The map() function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of map() function to double all the items in a list.

# Program to double each item in a list using map()

my\_list = [1, 5, 4, 6, 8, 11, 3, 12]

new\_list = list(map(lambda x: x \* 2 , my\_list))

print(new\_list)

Above Program Output below

[2, 10, 8, 12, 16, 22, 6, 24]

### 

### **Example**

A lambda function that adds **10** to the number passed in as an argument, and print the result:

x = lambda a : a + 10

print(x(5))

**Output:** 15

### Example

A lambda function that multiplies argument **a** with argument **b** and print the result:

x = lambda a, b : a \* b

print(x(5, 6))

**Output:** 30

### Example

A lambda function that sums argument a, b, and c and print the result:

x = lambda a, b, c : a + b + c

print(x(5, 6, 2))

**Output:** 13

**Summary of functions**

## **Types of Functions**

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

# **Python Built-in Functions**

# Python has several functions that are readily available for use. These functions are called built-in functions. Please refer back to the beginning of this reading.

# 

# 

# **Python User-defined Functions**

## **What are user-defined functions in Python?**

#### Functions that we define ourselves to do certain specific tasks are referred to as user-defined functions. The way in which we define and call functions in Python are already discussed above.

#### Functions that readily come with Python are called **built-in functions**. If we use functions written by others in the form of a library, it can be termed as library functions.

#### All the other functions that we write on our own fall under **user-defined functions**. So, our user-defined function could be a library function to someone else too.

## **Advantages of user-defined functions**

#### User-defined functions help to decompose a large program into small segments which makes the program easy to understand, maintain and debug.

#### If repeated code occurs in a program. A function can be used to include those codes and execute it when needed by calling that function.

#### Programmers working on large projects can divide the workload by making different functions.

**Example of a user-defined function**

# Program to illustrate

# the use of user-defined functions

def add\_numbers(x,y):

sum = x + y

return sum

num1 = 5

num2 = 6

print("The sum is", add\_numbers(num1, num2))

Above Program Output

Enter a number: 2.4

Enter another number: 6.5

The sum is 8.9

Here, we have defined the function **add\_numbers()** which adds two numbers and returns the result.

This is our user-defined function. We could have multiplied the two numbers inside our function (it's all up to us). But this operation would not be consistent with the name of the function. It would create ambiguity.

It is always a good idea to name functions according to the task they perform.

In the above example, **print()** is a built-in function in Python.

For your own exercise, could you please create a function get\_products() which will be able to multiply two numbers num1 and num1 and produce a product of the two.

## **Recursion**

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, tri\_recursion() is a function that we have defined to call itself ("recurse"). We use the k variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

### Example

Recursion Example

**def tri\_recursion(k):**

**if(k > 0):**

**result = k + tri\_recursion(k - 1)**

**print(result)**

**else:**

**result = 0**

**return result**

**print("\n\nRecursion Example Results")**

**tri\_recursion(6)**

### **How do you call functions in Python?**

Simply write the function's name followed by (), placing any required arguments within the brackets. For example, let's call the functions written above (in the previous example):

**# Define our 3 functions**

**def my\_function():**

**print("Hello From My Function!")**

**def my\_function\_with\_args(username, greeting):**

**print("Hello, %s , From My Function!, I wish you %s"%(username, greeting))**

**def sum\_two\_numbers(a, b):**

**return a + b**

**# print(a simple greeting)**

**my\_function()**

**#prints - "Hello, Nelson Mandela., From My Function!, I wish you a great year!"**

**my\_function\_with\_args("Nelson Mandela.", "2020 was a great year!")**

**# after this line x will hold the value 3!**

**x = sum\_two\_numbers(1,2)**

## **Exercise**

In this exercise you'll use an existing function, and while adding your own to create a fully functional program.

1. Add a function named list\_benefits() that returns the following list of strings: "More organized code", "More readable code", "Easier code reuse", "Allowing programmers to share and connect code together"
2. Add a function named build\_sentence(info) which receives a single argument containing a string and returns a sentence starting with the given string and ending with the string " is a benefit of functions!"
3. Run and see all the functions work together!

**# Modify this function to return a list of strings as defined above**

**def list\_benefits():**

**return "More organized code", "More readable code", "Easier code reuse", "Allowing programmers to share and connect code together"**

**# Modify this function to concatenate to each benefit - " is a benefit of functions!"**

**def build\_sentence(benefit):**

**return "%s is a benefit of functions!" % benefit**

**def name\_the\_benefits\_of\_functions():**

**list\_of\_benefits = list\_benefits()**

**for benefit in list\_of\_benefits:**

**print(build\_sentence(benefit))**

**name\_the\_benefits\_of\_functions()**

**Below is another version of the above code. Run both and see how they are different**

# Modify this function to return a list of strings as defined above

def list\_benefits():

pass

# Modify this function to concatenate to each benefit - " is a benefit of functions!"

def build\_sentence(benefit):

pass

def name\_the\_benefits\_of\_functions():

list\_of\_benefits = list\_benefits()

for benefit in list\_of\_benefits:

print(build\_sentence(benefit))

name\_the\_benefits\_of\_functions()