Functions in python(Part-1)

It is a block of code that is executed only when it is called. Functions help break down the programs into smaller chunks. These chunks of code are reusable and organised. They are also useful if you find yourself writing the same code at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

Introduction

Functions are defined with the def statement. The statement ends with a colon, and the code that is part of the function is indented below the def statement.

Difference between a Function and a Method

We discussed that a function contains a group of statements and performs a specific task. A function can be written individually in a Python program. A function is called using its name. When a function is written inside a class, it becomes a method. A method is called using one of the following ways:

objectname.methodname()
classname.methodname ()

So, remember that a function and a method are the same except their placement and the way they are called. In this chapter, we will learn how to create and use our own functions. Creating a function means defining a function or writing a function. Once a function is defined, it can be used by calling the function.

Defining a Function

We can define a function using the keyword def followed by function name. After the function name, we should write parentheses () which may contain parameters.For example,let's create a simple function that just prints something.

```
def firstPgm():
    print("Hello, World!")

firstPgm()
```

Output:

```
Hello, World!
```

Calling a Function

A function cannot run on its own. It runs only when we call it. So, the next step is to call the function using its name. While calling the function, we should pass the necessary values to the function in the parentheses as:

firstPgm()

Why do we need functions?

One use for functions is if you are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a function.

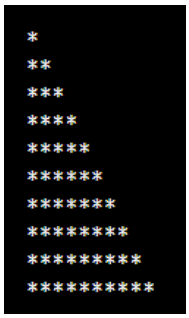
For instance, suppose for some reason you need to print a pyramid of stars like the one below at several points in your program.

Put the code into a function, and then whenever you need a pyramid, just call the function rather than typing several lines of redundant code. Here is the function.

```
def py():
    for i in range (1,11):
        for j in range(1, i+1):
            print("*", end="")
        print()

py()
```

Output:



One benefit of this is that if you decide to change the no of rows, you just have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

Returning Results from a Function

We can return the result or output from the function using a return' statement in the body of the function. For example,

```
return c # returns c value out of function
return 100 # returns 100
return lst # return the list that contains values
return x, y, c # returns 3 values
```

When a function does not return any result, we need not write the return statement in the body of the function. Now, we will rewrite our sum() function such that it will return the sum value rather than displaying it.

```
def sum(a, b):
    """This function finds sum of two numbers """
    c = a+b
    return c
    x=sum(10, 15)
    print('The sum is:' , x)
    y=sum(1.5, 10.75)
    print('The sum is: ', y)
```

In the above program, the result is returned by the sum() function through 'c' using the statement:

return c

When we call the function as:

x = sum(10, 15)

the result returned by the function comes into the variable ‘x'. Similarly, when we call the function as:

y=sum(1.5, 10.75)

The returned result will come into ‘y'.

Returning Multiple Values from a Function

A function returns a single value in the programming languages like C or Java. But in Python a function can return multiple values. When a function calculates multiple results and wants to return the results, we can use the return statement as return a, b, c

Here, three values which are in 'a', 'b', and 'c' are returned. These values are returned by the function as a tuple. Please remember a tuple is like a list that contains a group of elements To grab these values, we can use three variables at the time of calling the function as:

x, y, z = function()

Here, the variables ‘x’, ‘y' and 'z' are receiving the three values returned by the function. To understand this practically, we can create a function by the name sum_sub() that takes 2 values and calculates the results of addition and subtraction. These results are stored in the variables 'c' and 'd' and returned as a tuple by the function.

```
def sum_sub(a, b):
    c=a+b
    d=a-b
    return c,d
```

Since this function has two parameters, at the time of calling this function, we should pass two values, as:

```
x, y = sum_sub(10, 5)
```

Now, the result of addition which is in will be stored into ‘x' and the result of subtraction which is in 'd' will be stored into y.

```
def sum_sub(a, b):
    c=a+b
    d=a-b
    return c,d

x, y = sum_sub(10, 5)
print("Result of addition: ",x)
print("Result of subtraction: ",y)
```

Output:

```
Result of addition:  15
Result of subtraction:  5
```

Functions are First Class Objects

In Python, functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. Also, it is possible to return a function from another function. This is similar to returning an object (or value) from a function. The following possibilities are noteworthy:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as a parameter to another function.
- It is possible that a function can return another function.

To understand these points, we will take a simple program. In the program below, we have taken a function by the name display that returns a string. This function is called and the returned string is assigned to a variable 'x'.

A Python program to see how to assign a function to a variable.

```
def display (str):
    return 'Hi '+str

x = display("krishna")
print(x)
```

Output:

```
Hi krishna
```

Pass by Object Reference

In the languages like C and Java, when we pass values to a function, we think about two ways:

- Pass by value or call by value
- Pass by reference or call by reference

Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function. Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

Neither of these two concepts is applicable in Python. In Python, the values are sent to functions by means of object references. We know everything is considered as an object in Python. All numbers are objects, strings are objects, and the datatypes like tuples, lists, and dictionaries are also objects. If we store a value into a variable as:

x = 10

In this case, in other programming languages, a variable with name X is created and some memory is allocated to the variable. Then the value 10 is stored into the variable X. We can imagine ‘x' as a box where 10 is stored. This is not the case with Python. In Python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value 10 is created in memory for which a name 'x' is attached. So, 10 is the object and 'x' is the name or tag given to that object. Also, objects are created on heap

memory which is a very huge memory that depends on the RAM of our computer system. Heap memory is available during runtime of a program.

To know the location of an object in a heap, we can use the id() function that gives the identity number of an object. For example, consider the following code snippet

```
x = 10
id(x)
```

The preceding lines of code may display the following output:

1617805016

This number may change from computer to computer as it is computed depending on the available memory location where object '10' is stored in the computer. In general, two different objects will have different identity numbers.

When we pass values like numbers, strings, tuples or lists to a function, the references of these objects are passed to the function. To understand this concept, let's take a small program. In the program below, we are calling modify() function and passing x value, i.e. 10 to that function. Inside this function, we are modifying the value of ‘x' as 15. Inside the function, we are displaying 'x' value and its identity number, as:

```
print(x, id(x))
```

In the same manner, after coming out of the function, we are again displaying these values. Now see the program and its output.

```
def modify(x):
    x=15
    print(x, id(x))

x= 10
modify (x)
print(x, id(x))
```

Output:

```
15 9756672
10 9756512
```

Formal and Actual Arguments

When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called formal arguments. When we call the function, we should pass data or values to the function. These values are called actual arguments! In the following code, 'a' and ‘b' are formal arguments and ‘x' and 'y' are actual arguments.

```
def sum(a. b):
c=a+b
print (c)
x=10; y=15
sum(x, y)
```

The actual arguments used in a function call are of 4 types:

- Positional arguments
- Keyword arguments
- Default arguments
- Variable length arguments

Positional Arguments

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call For example, take a function definition with two arguments as:

def attach(s1, s2)
This function expects two strings that too in that order only. Let's assume that this function attaches the two strings as s1+s2. So, while calling this function, we are supposed to pass only two strings as:

```
attach("New", "York')
```

The preceding statement displays the following output:

```
NewYork
```

Suppose, we passed ‘York' first and then ‘New', then the result will be: ‘YorkNew'. Also, if we try to pass more than or less than 2 strings, there will be an error. For example, if we call the function by passing 3 strings as:

attach(‘New’, ‘York', City')
Then there will be an error displayed.

A Python program to understand the positional arguments of a function.

```
def attach(s1, s2):

s3=s1+s2
print("Total string: “+s3)

attach('New', 'York')
```

Keyword Arguments

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

```
def grocery(item, price):
```

At the time of calling this function, we have to pass two values and we can mentsm which value is for what. For example,

```
grocery (item="Sugar”, price=50.75)
```

Here, we are mentioning a keyword 'item' and its value and then another keyword 'price' and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as

```
grocery (price-88.00, item-’oil’)
```
In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value. The program below demonstrates how to use keyword arguments while calling grocery() function.

```
def grocery (item, price):
    print('Item= %s' % item)
    print('Price = %.2f' % price)

grocery (item='Sugar', price=50.75)
grocery (price=88.00, item='oil')
```

Output:

```
Item= Sugar
Price = 50.75
Item= oil
Price = 88.00
```

Default Arguments

We can mention some default values for the function parameters in the definition. Let's take the definition of grocery() function as:

def grocery (item, price=40.00):

Here, the first argument is' item' whose default value is not mentioned. But the second argument is 'price' and its default value is mentioned to be 40.00. At the time of calling this function, if we do not pass 'price' value, then the default value of 40.00 is taken. If we mention the 'price' value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The program below will clarify this.
A Python program to understand the use of default functions.

```
def grocery(item, price=40.00):
    print('Item= %s' % item)
    print("Price=%.2f"  %price)

grocery(item='Sugar', price=50.75)
grocery(item='Sugar')
```

Output:

```
Item= Sugar
Price=50.75
Item= Sugar
Price=40.00
```

Variable Length Arguments

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. For example, if the programmer is writing a function to add the numbers, we can write:

add(a, b)

But, the user who is using this function may want to use this function to find the sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

add (10, 15, 20)

Then the add() function will fail and an error will be displayed. If the programmer wants to develop a function that can accept 'n' arguments, that is also possible in Python. For that purpose, a variable length argument is used in the function definition. A variable length argument is an argument that can accept any number of values. The variable lengt/h argument is written with a symbol before it in the function definition as:

def add(farg, *args):

Here, 'farg' is the formal argument and ‘*args' represents variable length arguments. We can pass I or more values to this "args' and it will store them all in a tuple. A tuple is like a list where a group of elements can be stored. In the program below, we are showing how to use variable length arguments.

A Python program to show variable length arguments and its use.

```
def add(farg, *args):
    print('Formal argument= ', farg)
    sum=0
    for i in args:
        sum+=i
        print('Sum of all numbers= ', (farg+sum))

add(5,10)
add(5,10,20,30)
```

Output:

```
Formal argument=  5
Sum of all numbers=  15
Formal argument=  5
Sum of all numbers=  15
Sum of all numbers=  35
Sum of all numbers=  65
```

Local and Global Variables

When we declare a variable inside a function, it becomes a local variable. A local variable in a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function. In the following example, the variable 'a' is declared inside myfunction() and hence it is available inside that function. Once we come out of the function, the variable is removed from memory and it is not available. Consider the following code:

```
def myfunction():
    a=1
    a+=1
    print(a)

myfunction()
print(a)
```

Output:

```
Traceback (most recent call last):
  File "./prog.py", line 7, in <module>
NameError: name 'a' is not defined
```

See the last statement where we are displaying 'a' value outside the function. This statement raises an error with a message: name 'a' is not defined.

When a variable is declared above a function, it becomes a global variable. Such variables are available to all the functions which are written after it. Consider the following code:

```
a=1
def myfunction():
    b=2
    print('a= ',a)
    print('b= ',b)

myfunction()
print(a)
print (b)
```

Output:

```
Traceback (most recent call last):
  File "./prog.py", line 9, in <module>
NameError: name 'b' is not defined
```

Whereas the scope of the local variable is limited only to the function where it is declared, the scope of the global variable is the entire program body written below it.

The Global Keyword

Sometimes, the global variable and the local variable may have the same name. In thi case, the function, by default, refers to the local variable and ignores the global variable So, the global variable is not accessible inside the function but outside of it, it is accessible. Consider the program below.

A Python program to understand global and local variables.

```
a=1
def myfunction():
    a=2
    print('a= ',a)
myfunction()
print('a= ',a)
```

Output:

```
a=  2
a=  1
```

Namespaces and Scope in Python

Namespaces:

Roughly speaking, namespaces are just containers for mapping names to objects. As you might have already heard, everything in Python - literals, lists, dictionaries, functions, classes, etc. - is an object.
Such a “name-to-object” mapping allows us to access an object by a name that we’ve assigned to it. E.g.,If we make a simple string assignment via a_string = "Hello string", we create a reference to the "Hello string" object, and henceforth we can access it via its variable name a_string.
We can picture a namespace as a Python dictionary structure, where the dictionary keys represent the names and the dictionary values the object itself (and this is also how namespaces are currently implemented in Python), e.g.,

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
```

Now, the tricky part is that we have multiple independent namespaces in Python, and names can be reused for different namespaces (only the objects are unique, for example:

```
a_namespace = {'name_a':object_1, 'name_b':object_2, ...}
b_namespace = {'name_a':object_3, 'name_b':object_4, ...}
```

For example, every time we call a for-loop or define a function, it will create its own namespace. Namespaces also have different levels of hierarchy (the so-called “scope”), which we will discuss in more detail in the next section.

Scope:

In the section above, we have learned that namespaces can exist independently from each other and that they are structured in a certain hierarchy, which brings us to the concept of “scope”. The “scope” in Python defines the “hierarchy level” in which we search namespaces for certain “name-to-object” mappings.
For example, let us consider the following code:

```
i = 1
def pgm():
    i = 5
    print(i, 'in pgm()')

print(i, 'global')

pgm()
```

Output:

```
1 global
5 in pgm()
```

Here, we just defined the variable name i twice, once on the pgm function.

- pgm_namespace = {'i':object_3, ...}
- global_namespace = {'i':object_1, 'name_b':object_2, ...}

Example of Scope and Namespace in Python:

```
def outer_function():
    b = 20
    def inner_func():
        c = 30

a = 10
```

Here, the variable a is in the global namespace. Variable b is in the local namespace of outer_function() and c is in the nested local namespace of inner_function().

When we are in inner_function(), c is local to us, b is non-local and a is global. We can read as well as assign new values to c but can only read b and a from inner_function().

If we try to assign a value to b, a new variable b is created in the local namespace which is different from the non-local b. The same thing happens when we assign a value to a.

However, if we declare a as global, all the reference and assignment go to the global a. Similarly, if we want to rebind the variable b, it must be declared as non-local. The following example will further clarify this.

```
def outer_function():
    a = 20

    def inner_function():
        a = 30
        print('a =', a)

    inner_function()
    print('a =', a)


a = 10
outer_function()
print('a =', a)
```

Output:

```
a = 30
a = 20
a = 10
```

In this program, three different variables ‘a’ are defined in separate namespaces and accessed accordingly. While in the following program,

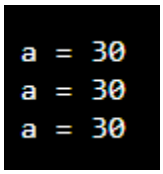
```
def outer_function():
    global a
    a = 20

    def inner_function():
        global a
        a = 30
        print('a =', a)

    inner_function()
    print('a =', a)


a = 10
outer_function()
print('a =', a)
```

Output:



Here, all references and assignments are to the global a due to the use of keyword global.