Python - User-Defined Functions

Python includes many built-in functions. These functions perform a predefined task and can be called upon in any program, as per requirement. However, if you don't find a suitable built-in function to serve your purpose, you can define one. We will now see how to define and use a function in a Python program.

Defining a Function

A function is a reusable block of programming statements designed to perform a certain task. To define a function, Python provides the def keyword. The following is the syntax of defining a function.

Syntax:

```
def function_name(parameters):
   "function docstring"
   statement1
   statement2
   ...
   ...
   return [expr]
```

The keyword def is followed by a suitable identifier as the name of the function and parentheses. One or more parameters may be optionally mentioned inside parentheses. The : symbol after parentheses starts an indented block.

The first statement in this block is an explanatory string which tells something about the functionality. It is called a docstring and it is optional. It is somewhat similar to a comment. Subsequent statements that perform a certain task form the body of the function.

The last statement in the function block includes the return keyword. It sends an execution control back to calling the environment. If an expression is added in front of return, its value is also returned.

Given below is the definition of the SayHello() function. When this function is called, it will print a greeting message.

Example: Function

```
def SayHello():
   ""First line is docstring. When called, a greeting message will be displayed""
   print ("Hello! Welcome to Python tutorial on Altran's")
   return
```

To call a defined function, just use its name as a statement anywhere in the code. For example, the above function can be called as SayHello() and it will show the following output.

```
>>> SayHello()
Hello! Welcome to Python tutorial on Altran's
```

Function with Parameters

It is possible to define a function to receive one or more parameters (also called arguments) and use them for processing inside the function block. Parameters/arguments may be given suitable formal names. The SayHello() function is now defined to receive a string parameter called name. Inside the function, print() statement is modified to display the greeting message addressed to the received parameter.

Example: Parameterized Function

```
def SayHello(name):
   print ("Hello {}!.".format(name))
   return
```

You can call the above function as shown below.

```
>>> SayHello("Gandhi")
Hello Gandhi!
```

The names of the arguments used in the definition of the function are called formal arguments/parameters. Objects actually used while calling the function are called actual arguments/parameters.

In the following example, the result() function is defined with three arguments as marks. It calculates the percentage and displays pass/fail result. The function is called by providing different values on every call.

FunctionTest.py

```
def result(m1,m2,m3):
    ttl=m1+m2+m3
    percent=ttl/3
    if percent>=50:
        print ("Result: Pass")
    else:
        print ("Result: Fail")
    return

p=int(input("Enter your marks in physics: "))
c=int(input("Enter your marks in chemistry: "))
m=int(input("Enter your marks in maths: "))
result(p,c,m)
```

Run the above script in IDLE with two different sets of inputs is shown below:

Result:

```
Enter your marks in physics: 50
Enter your marks in chemistry: 60
Enter your marks in maths: 70
Result: Pass

Enter your marks in physics: 30
Enter your marks in chemistry: 40
Enter your marks in maths: 50
Result: Fail
```

Parameter with Default Value

While defining a function, its parameters may be assigned default values. This default value gets substituted if an appropriate actual argument is passed when the function is called. However, if the actual argument is not provided, the default value will be used inside the function.

The following SayHello() function is defined with the name parameter having the default value 'Guest'. It will be replaced only if some actual argument is passed.

Example: Parameter with Default Value

```
def SayHello(name='Guest'):
   print ("Hello " + name)
   return
```

You can call the above function with or without passing a value, as shown below.

```
>>> SayHello()
Hello Guest
>>> SayHello('Tom')
Hello Tom
```

Function with Keyword Arguments

In order to call a function with arguments, the same number of actual arguments must be provided. Consider the following function.

```
def AboutMe(name, age):
   print ("Hi! My name is {} and I am {} years old".format(name,age))
   return
```

The above function is defined to receive two positional arguments. If we try to call it with only one value passed, the Python interpreter flashes TypeError with the following message:

```
>>> AboutMe("Ramesh")
TypeError: AboutMe() missing 1 required positional argument: 'age'
```

If a function receives the same number of arguments and in the same order as defined, it behaves correctly.

```
>>> AboutMe("Mohan", 23)
Hi! My name is Mohan and I am 23 years old
```

Python provides a useful mechanism of using the name of the formal argument as a keyword in function call. The function will process the arguments even if they are not in the order provided in the definition, because, while calling, values are explicitly assigned to them. The following calls to the `AboutMe()` function is perfectly valid.

```
>>> AboutMe(age=23, name="Mohan")
Hi! My name is Mohan and I am 23 years old
```

Function with Return Value

Most of the times, we need the result of the function to be used in further processes. Hence, when a function returns, it should also return a value.

A user-defined function can also be made to return a value to the calling environment by putting an expression in front of the return statement. In this case, the returned value has to be assigned to some variable.

Consider the following example function with the return value.

Example: Function with Return Value

```
def sum(a, b):
   return a + b
```

The above function can be called and provided the value, as shown below.

```
>>> total=sum(10, 20)
>>> total
30
>>> total=sum(5, sum(10, 20))
>>> total
35
```

Passing Arguments by Reference

In Python, arguments are always passed by reference. The following snippet will confirm this:

Example: Passing Argument by Reference

```
def myfunction(arg):
   print ('value received:',arg,'id:',id(arg))
   return
x=10
print ('value passed:',x, 'id:',id(x))
myfunction(x)
```

The built-in id() function returns a unique integer corresponding to the identity of an object. In the above code, id() of x before and after passing to a function shows a similar value.

```
>>> x=10
>>> id(x)
1678339376
>>> myfunction(x)
value received: 10 id: 1678339376
```

If we change the above number variable inside a function then it will create a different variable as number, which is immutable. However, if we modify a mutable list object inside the function, and display its contents after the function is completed, the changes are reflected outside the function as well.

Example: Passing List by Reference

```
def myfunction (list):
   list.append(40)
   print ("Modified list inside a function: ", list)
   return
```

The following result confirms that arguments are passed by reference to a Python function.

```
>>> mylist=[10,20,30]
>>> myfunction(mylist)
Modified list inside a function: [10, 20, 30, 40]
>>> mylist
[10, 20, 30, 40]
```