PYTHON PROGRAMMING - I

Chap - 5

Python Functions And Modules

By-

Prof. A. P. Chaudhari

(M.Sc. Computer Science, SET)

HOD,

Department of Computer Science

S.V.S's Dadasaheb Rawal College, Dondaicha

Functions:

- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- A function is a block of organized, reusable code that is used to perform a single, related action.
- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- Furthermore, it avoids repetition and makes the code reusable.
- As you already know, Python gives you many built-in functions like print, etc. but you can also create your own functions. These functions are called *user-defined functions.*

Functions:

Defining a Function:

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon : and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Functions:

Syntax:

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example:

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Functions:

Calling a Function:

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme function −

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

O/P:- I'm first call to user defined function!
Again second call to the same function

Functions:

Pass by reference vs value:

All parameters arguments in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
# Function definition is here
      def changeme( mylist ):
         "This changes a passed list into this function"
         mylist.append([1,2,3,4]);
         print "Values inside the function: ", mylist
         return
# Now you can call changeme function
      mylist = [10,20,30];
      changeme( mylist );
      print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
      Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
      Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

Functions:

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here
        def changeme( mylist ):
           "This changes a passed list into this function"
           mylist = [1,2,3,4];          # This would assign new reference in mylist
           print "Values inside the function: ", mylist
           return
 # Now you can call changeme function
        mylist = [10,20,30];
        changeme( mylist );
        print "Values outside the function: ", mylist
```

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

```
        Values inside the function:  [1, 2, 3, 4]
        Values outside the function:  [10, 20, 30]
```

Functions:

Function Arguments:

You can call a function by using the following types of formal arguments −

1) Required arguments
2) Keyword arguments
3) Default arguments
4) Variable-length arguments

1) Required arguments:

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

Functions:

To call the function *printme*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
# Function definition is here
        def printme( str ):
           "This prints a passed string into this function"
           print str
           return;
# Now you can call printme function
        printme()
```

O/P:-

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

Functions:

2) Keyword arguments:

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme* function in the following ways −

```
# Function definition is here
      def printme( str ):
         "This prints a passed string into this function"
         print str
         return;
 # Now you can call printme function
      printme( str = "My string")
```

O/P:- My string

Functions:

The following example gives more clear picture. Note that the order of parameters does not matter.

```
# Function definition is here
    def printinfo( name, age ):
       "This prints a passed info into this function"
       print "Name: ", name
       print "Age ", age
       return;
# Now you can call printinfo function
    printinfo( age=50, name="miki" )
```

O/P:-

```
Name:  miki
Age  50
```

Functions:

3) Default arguments:

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 following example gives an idea on default arguments, it prints default age if it is not passed −

```
# Function definition is here
    def printinfo( name, age = 35 ):
       "This prints a passed info into this function"
       print "Name: ", name
       print "Age ", age
       return;
 # Now you can call printinfo function
    printinfo( age=50, name="miki" )
    printinfo( name="miki" )
```

O/P:-

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Functions:

4) Variable-length arguments:

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk ∗ is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Functions:

Following is a simple example −

```
# Function definition is here
    def printinfo( arg1, *vartuple ):
       "This prints a variable passed arguments"
       print "Output is: "
       print arg1
       for var in vartuple:
          print var
       return;
# Now you can call printinfo function
    printinfo( 10 )
    printinfo( 70, 60, 50 )
```

O/P:-

```
Output is:
10
Output is:
70
60
50
```

Functions:

The *Anonymous* Functions:

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++.

Functions:

Syntax:

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]] : expression

Following is the example to show how *lambda* form of function works −

```
# Function definition is here
    sum = lambda arg1, arg2: arg1 + arg2;
# Now you can call sum as a function
    print "Value of total : ", sum( 10, 20 )
    print "Value of total : ", sum( 20, 20 )
```

O/P:-

```
Value of total :  30
Value of total :  40
```

Functions:

The *return* Statement:

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
# Function definition is here
      def sum( arg1, arg2 ):
         "Add both the parameters and return them."
         total = arg1 + arg2
         print "Inside the function : ", total
         return total;
# Now you can call sum function
      total = sum( 10, 20 );
      print "Outside the function : ", total
```

O/P:-

```
Inside the function :  30
Outside the function :  30
```

Functions:

Scope of Variables:

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable. The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Functions:

Following is a simple example −

```
total = 0;              # This is global variable.
# Function definition is here
    def sum( arg1, arg2 ):
      "Add both the parameters and return them."
      total = arg1 + arg2;              # Here total is local variable.
      print "Inside the function local total : ", total
      return total;
# Now you can call sum function
    sum( 10, 20 );
    print "Outside the function global total : ", total
```

O/P:-

```
Inside the function local total :  30
Outside the function global total :  0
```

Module:

A module is to be the same as a code library. A file containing a set of functions you want to include in your application. A module is a file containing Python definitions and statements. A module can define functions, classes and variables.

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Creating Module: To create a module just save the code you want in a file with file extension .py

e.g: Write the following code and save file as **module1.py**

```
def  fun1(name):
        print ‘Hello ’, name
        return
```

Module:

How to use module:

You can use any Python source file as a module by executing an **import statement** in some other Python source file.

Syntax − import module1[, module2,... moduleN]

When the interpreter encounters an import statement, it imports the module.

For example, to import the module **module1.py**, you need to put the following command at the top of the script −

```
# Import module module1
import module1
# Now you can call defined function that module as follows
module1.fun1(“Parth")
```

O/P: Hello Parth

Module:

Variables in module:

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc).

e.g.: Following code save in module1.py file.

```
person1 = {'Name':'Hitesh', 'Age':25, 'City':'Pune'}
```

Import the module named module1, and access the person1 dictionary:

```
import module1
a = z1.person1['Age']
print 'Age:', a
```

O/P: Age: 25

Module:

Built-in Modules:

A large number of pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in modules. A module is a file containing definitions of functions, classes, variables or any other Python objects. Contents of this file can be made available to any other program.

There are several built-in modules in Python, which you can import whenever you like. They are loaded automatically as the interpreter starts and are always available.

e.g.: import platform

```
a = platform.system()
print a
```

O/P: Windows

Module:

1) OS Module:

It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.

i) Creating Directory:

We can create a new directory using the **mkdir()** function from the OS module.

e.g.:
```
import os
os.mkdir("D:\Demo1")
```

A new directory corresponding to the path in the string argument of the function will be created. If we open D drive, we should notice Demo1 folder created.

Module:

ii) Fetching contents from directory:

Python method **listdir()** returns a list containing the names of the entries in the directory given by path.

e.g.:
```
import os
files = os.listdir("D:\ICT")
for f in files:
    print f
```

O/P:
```
01 Syllabus
02 Reference Book
03 TrainingPPTs Sessionwise
04 WorkBook
05 TrainingSoftwares
untitled.bmp
```

Module:

iii) Remove Directory:

Python method **removedirs()** removes directories recursively. If the leaf directory is successfully removed, removedirs() tries to successively remove every parent directory displayed in path.

e.g.:

```
import os
os.removedirs("D:\Demo1")
```

Module:

2) Math module:

Python math module is defined as the most famous mathematical functions, which includes trigonometric functions, representation functions, logarithmic functions, etc. Furthermore, it also defines two mathematical constants, i.e., Pie and Euler number, etc.

Pie (n): It is a well-known mathematical constant and defined as the ratio of circumstance to the diameter of a circle. Its value is 3.141592653589793.

e.g.: import math

print(math.pi) O/P: 3.14159265359

Euler's number(e): It is defined as the base of the natural logarithmic, and its value is 2.718281828459045.

e.g.: import math

print(math.e) O/P: 2.71828182846

Module:

i) log10(): This method returns base 10 logarithm of the given number and called the standard logarithm.

e.g.:
```
import math
x=13
print 'log10(13) is :', math.log10(x)
```
O/P: log10(13) is : 1.11394335231

ii) pow(x,y): This method returns the power of the x corresponding to the value of y. If value of x is negative or y is not integer value than it raises a ValueError.

e.g.:
```
import math
number = math.pow(10,2)
print "The power of number:",number
```
O/P: The power of number: 100.0

Module:

iii) floor(x): This method returns the floor value of the x. It returns the less than or equal value to x.

e.g.:
```
import math
number = math.floor(10.25201)
print "The floor value is:",number
```
O/P: The floor value is: 10.0

iv) ceil(x): This method returns the ceil value of the x. It returns the greater than or equal value to x.

e.g.:
```
import math
number = math.ceil(10.25201)
print "The Ceilling value is:",number
```
O/P: The Ceilling value is: 11.0