Chapter 2

Functions

After studying this lesson, students will be able to:

- *Understand and apply the concept of module programming*
- *Write functions*
- *Identify and invoke appropriate predefined functions*
- *Create Python functions and work in script mode.*
- *Understand arguments and parameters of functions*
- *Work with different types of parameters and arguments*
- *Develop small scripts involving simple calculations*

Introduction

Remember, we earlier talked about working in script mode in chapter-1 of this unit to retain our work for future usage. For working in script mode, we need to *write a function* in the Python and save it in the file having **.py** extension.

A function is a named sequence of statement(s) that performs a computation. It contains line of code(s) that are executed sequentially from top to bottom by Python interpreter. They are the most important building blocks for any software in Python.

Functions can be categorized as belonging to

i. Modules

ii. Built in

iii. User Defined

Module

A module is a file containing Python definitions (i.e. functions) and statements. Standard library of Python is extended as module(s) to a programmer. Definitions from the module can be used within the code of a program. To use these modules in the program, a programmer needs to import the module. Once you import a module, you can reference (use), any of its functions or variables in your code. There are many ways to import a module in your program, the one's which you should know are:

i. import

ii. from

Import

It is simplest and most common way to use modules in our code. Its syntax is:

import modulename1 [,modulename2, ---------]

Example

```
>>> import math
```

On execution of this statement, Python will

(i) search for the file '**math.py**'.

(ii) Create space where modules definition & variable will be created,

(iii) then execute the statements in the module.

Now the definitions of the module will become part of the code in which the module was imported.

To use/ access/invoke a function, you will specify the module name and name of the function- separated by dot (.). This format is also known as *dot notation*.

Example

```
>>> value= math.sqrt (25)          # dot notation
```

The example uses sqrt( ) function of module **math** to calculate square root of the value provided in parenthesis, and returns the result which is inserted in the ***value***. The expression (variable) written in parenthesis is known as argument (actual argument). It is common to say that the function takes arguments and return the result.

This statement invokes the sqrt ( ) function. We have already seen many function invoke statement(s), such as

```
>>> type ( )

>>> int ( ), etc.
```

From Statement

It is used to get a specific function in the code instead of the complete module file. If we know beforehand which function(s), we will be needing, then we may use **from**. For modules having large no. of functions, it is recommended to use **from** instead of import. Its syntax is

>>> from modulename import functionname [, functionname…..]

Example

```
>>> from math import sqrt

value = sqrt (25)
```

Here, we are importing sqrt function only, instead of the complete math module. Now sqrt( ) function will be directly referenced to. These two statements are equivalent to previous example.

```
from modulename import *
```

will import everything from the file.

Note: You normally put all import statement(s) at the beginning of the Python file but technically they can be anywhere in program.

Lets explore some more functions available in **math module:**

Name of the function	Description	Example

ceil(x)	It returns the smallest integer not less than x, where *x is a numeric expression.*	math.ceil(-45.17)
		-45.0
		math.ceil(100.12)
		101.0
		math.ceil(100.72)
		101.0

floor(x)	It returns the largest integer not greater than x, where *x is a numeric expression.*	math.floor(-45.17)
		-46.0
		math.floor(100.12)
		100.0
		math.floor(100.72)
		100.0

fabs(x)	It returns the absolute value of x, *where x is a numeric value.*	math.fabs(-45.17)
		45.17
		math.fabs(100.12)
		100.12
		math.fabs(100.72)
		100.72

exp(x)	It returns exponential of x: e^x, where x is a numeric expression.	math.exp(-45.17)
		2.41500621326e-20
		math.exp(100.12)
		3.03084361407e+43
		math.exp(100.72)
		5.52255713025e+43

log(x)	It returns natural logarithm of x, for x > 0, where *x is a numeric expression.*	math.log(100.12)
		4.60636946656
		math.log(100.72)
		4.61234438974

log10(x)	It returns base-10 logarithm of x for x > 0, where *x is a numeric expression.*	math.log10(100.12)
		2.00052084094
		math.log10(100.72)
		2.0031157171

pow(x, y)	It returns the value of x^y, *where x and **y** are numeric expressions.*	math.pow(100, 2)
		10000.0
		math.pow(100, -2)

		0.0001 math.pow(2, 4) **16.0** math.pow(3, 0) **1.0**

sqrt (x)	It returns the square root of x for x > 0, where x is a numeric expression.	math.sqrt(100)
		10.0
		math.sqrt(7)
		2.64575131106

cos (x)	It returns the cosine of x in radians, *where x is a numeric expression*	math.cos(3)
		-0.9899924966
		math.cos(-3)
		-0.9899924966
		math.cos(0)
		1.0
		math.cos(math.pi)
		-1.0

sin (x)	It returns the sine of x, in radians, *where x must be a numeric value.*	math.sin(3)
		0.14112000806
		math.sin(-3)
		-0.14112000806
		math.sin(0)
		0.0

tan (x)	It returns the tangent of x in radians, *where x must be a numeric value.*	math.tan(3)
		-0.142546543074
		math.tan(-3)
		0.142546543074
		math.tan(0)
		0.0

degrees (x)	It converts angle x from radians to degrees, *where x must be a numeric value.*	math.degrees(3)
		171.887338539
		math.degrees(-3)
		-171.887338539
		math.degrees(0)
		0.0

radians(x)	It converts angle x from degrees to radians, *where x must be a numeric value.*	math.radians(3)
		0.0523598775598
		math.radians(-3)
		-0.0523598775598
		math.radians(0)
		0.0

Some functions from **random module** are:

Name of the function	Description	Example
random ( )	It returns a random float x, such that 0 ≤ x<1	>>>random.random ( ) **0.281954791393** >>>random.random ( ) **0.309090465205**

randint (a, b)	It returns a int x between a & b such that
	a ≤ x ≤ b
		>>> random.randint (1,10)
		5
		>>> random.randint (-2,20)
		-1

uniform (a,b)	It returns a floating point number x, such that a <= x < b	>>>random.uniform (5, 10) **5.52615217015**

randrange ([start,] stop [,step])	It returns a random item from the given range	>>>random.randrange(100,1000,3) **150**

Some of the other modules, which you can explore, are: string, time, date

Built in Function

Built in functions are the function(s) that are built into Python and can be accessed by a programmer. These are always available and for using them, we don't have to import any module (file). Python has a small set of built-in functions as most of the functions have been partitioned to modules. This was done to keep core language precise.

Name	Description	Example

abs (x)	It returns distance between x and zero, where *x is a numeric expression.*	>>>abs(-45)
		45
		>>>abs(119L)
		119

max(x, y, z, ...)	It returns the largest of its arguments: where x, y and z are numeric variable/expression.	>>>max(80, 100, 1000)
		1000
		>>>max(-80, -20, -10)
		-10

min(x, y, z, ...)	It returns the smallest of its arguments; where x, y, and z are numeric variable/expression.	>>> min(80, 100, 1000)
		80
		>>> min(-80, -20, -10)
		-80

cmp(x, y)	It returns the sign of the difference of two numbers: -1 if x < y, 0 if x == y, or 1 if x > y, *where x and y are numeric variable/expression.*	>>>cmp(80, 100)
		-1
		>>>cmp(180, 100)
		1

divmod (x,y)	Returns both quotient and remainder by division through a tuple, when x is divided by y; where x & y are variable/expression.	>>> divmod (14,5)
		(2,4)
		>>> divmod (2.7, 1.5)
		(1.0, 1.20000)

len (s)	Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).	>>> a= [1,2,3]
		>>>len (a)
		3
		>>> b= 'Hello'
		>>> len (b)
		5

range (*start*, *stop*[, *step*])	This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If *step* is positive, the last element is the largest start + i * step less than *stop*; if *step* is negative, the last element is the smallest start + i * step greater than *stop*. *step* must not be zero (or else Value Error is raised).	>>> range(10)
		[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
		>>> range(1, 11)
		[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		>>> range(0, 30, 5)
		[0, 5, 10, 15, 20, 25]
		>>> range(0, 10, 3)
		[0, 3, 6, 9]
		>>> range(0, -10, -1)
		[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
		>>> range(0)
		[]
		>>> range(1, 0)
		[]

round(x [, n])	It returns float x rounded to n digits from the decimal point, *where x and n are numeric expressions.* If n is not provided then x is rounded to 0 decimal digits.	>>>round(80.23456, 2) **80.23** >>>round(-100.000056, 3) **-100.0** >>> round (80.23456) **80.0**

Apart from these functions, you have already seen the use of the following functions:

bool ( ), chr ( ), float ( ), int ( ), long (), str ( ), type ( ), id ( ), tuple ( )

Composition

Composition is an art of combining simple function(s) to build more complicated ones, i.e., result of one function is used as the input to another.

Example

Suppose we have two functions fn1 & fn2, such that

```
a= fn2 (x)

b= fn1 (a)
```

then call to the two functions can be combined as

```
b= fn1 (fn2 (x))
```

Similarly, we can have statement composed of more than two functions. In that result of one function is passed as argument to next and result of the last one is the final result.

Example

```
math.exp (math.log (a+1))
```

Example

```
degrees=270

math.sin (degrees/360.0 *2*math.pi)
```

Composition is used to package the code into modules, which may be used in many different unrelated places and situations. Also it is easy to maintain the code.

Note: Python also allow us to take elements of program and compose them.

User Defined Functions

So far we have only seen the functions which come with Python either in some file (module) or in interpreter itself (built in), but it is also possible for programmer to write their own function(s). These functions can then be combined to form a module which can then be used in other programs by importing them.

To define a function keyword **def** is used. After the keyword comes an identifier i.e. name of the function, followed by parenthesized list of parameters and the colon which ends up the line. Next follows the block of statement(s) that are the part of function.

Before learning about Function header & its body, lets explore block of statements, which become part of function body.

Block of statements

A block is one or more lines of code, grouped together so that they are treated as one big sequence of statements while executing. In Python, statements in a block are written with indentation. Usually, a block begins when a line is indented (by four spaces) and all the statements of the block should be at same indent level. A block within block begins when its first statement is indented by four space, i.e., in total eight spaces. To end a block, write the next statement with the same indentation before the block started.

Now, lets move back to function- the **Syntax** of function is:

```
def NAME ([PARAMETER1, PARAMETER2, …..]): #Square brackets include
statement(s)                              #optional part of statement
```

Let's write a function to greet the world:

```
def sayHello ():                # Line No. 1
    print "Hello World!"        # Line No.2
```

The first line of function definition, i.e., Line No. 1 is called **header** and the rest, i.e. Line No. 2 in our example, is known as **body**. Name of the function is ***sayHello***, and empty parenthesis indicates no parameters. Body of the function contains one Python statement, which displays a string constant on screen. So the general structure of any function is

Function Header

It begins with the keyword def and ends with colon and contains the function identification details. As it ends with colon, we can say that what follows next is, block of statements.

Function Body

Consisting of sequence of indented (4 space) Python statement(s), to perform a task.

Defining a function will create a variable with same name, but does not generate any result. The body of the function gets executed only when the function is called/invoked. Function **call** contains the name of the function (being executed) followed by the list of values (i.e. arguments) in parenthesis. These arguments are assigned to parameters from LHS.

```
>>> sayHello ()          # Call/invoke statement of this function
```

Will produce following on screen

```
Hello World!
```

Apart from this, you have already seen many examples of invoking of functions in Modules & Built-in Functions.

Let's know more about **def.** It is an executable statement. At the time of execution a function is created and a name (name of the function) is assigned to it. Because it is a statement, **def** can appear anywhere in the program. It can even be nested.

Example

```
if condition:
    def fun ( ):                # function definition one way
.
```

```
.
.
else:
        def fun ( ):                      # function definition other way
.
.
.
fun ( )                           #  calls the function selected.
```

This way we can provide an alternative definition to the function. This is possible because def is evaluated when it is reached and executed.

```
def fun (a):
```

Let's explore Function body

The first statement of the function body can optionally be a string constant, **docstring,** enclosed in triple quotes. It contains the essential information that someone might need about the function, such as

- What function does (**without How it does**) i.e. summary of its purpose
- Type of parameters it takes
- Effect of parameter on behavior of functions, etc.

DocString is an important tool to document the program better, and makes it easier to understand. We can actually access docstring of a function using __ doc__ (function name). Also, when you used help(), then Python will provide you with docstring of that function on screen. So it is strongly recommended to use docstring … when you write functions.

Example

```
def area (radius):
        """ calculates area of a circle.                          docstring begins
        require an integer or float value to calculate area.
        returns the calculated value to calling function """      docstring ends
        a=radius**2
        return a
```

Function is pretty simple and its objective is pretty much clear from the docString added to the body.

The last statement of the function, i.e. return statement returns a value from the function. Return statement may contain a constant/literal, variable, expression or function, if return is used without anything, it will return **None.** In our example value of a variable **area** is returned.

Instead of writing two statements in the function, i.e.

```
        a = radius **2
        return a
```

We could have written

```
        return radius **2
```

Here the function will first calculate and then return the value of the expression.

It is possible that a function might not return a value, as sayHello( ) was not returning a value. sayHello( ) prints a message on screen and does not contain a return statement, such functions are called **void functions**.

Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result of such function to a variable, you get a special value called **None**.

Example

```
        def check (num):
                if (num%2==0):
                print "True"
      else:
                print "False"
>>> result = check (29)
```

```
False

>>> print result

None
```

DocString Conventions:

- ✡ The first line of a docstring starts with capital letter and ends with a period (.)
- ✡ Second line is left blank (it visually separates summary from other description).
- ✡ Other details of docstring start from 3rd line.

Parameters and Arguments

Parameters are the value(s) provided in the parenthesis when we write function header. These are the values required by function to work. Let's understand this with the help of function written for calculating area of circle.

radius is a parameter to function area.

If there is more than one value required by the function to work on, then, all of them will be listed in parameter list separated by comma.

Arguments are the value(s) provided in function call/invoke statement. List of arguments should be supplied in same way as parameters are listed. Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.

Example

of argument in function call

```
>>> area (5)
```

5 is an argument. An argument can be constant, variable, or expression.

Scope of Variables

Scope of variable refers to the part of the program, where it is visible, i.e., area where you can refer (use) it. We can say that scope holds the current set of variables and their values. We will study two types of scope of variables- global scope or local scope.

Global Scope

A variable, with global scope can be used anywhere in the program. It can be created by defining a variable outside the scope of any function/block.

Example

```
x=50
def test ( ):
    print "Inside test x is" , x
print "Value of x is" , x
```

on execution the above code will produce

```
Inside test x is 50
Value of x is 50
```

Any modification to global is permanent and visible to all the functions written in the file.

Example

```
x=50
def test ( ):
    x+= 10
    print "Inside test x is", x
print "Value of x is", x
```

will produce

```
Inside test x is 60
Value of x is 60
```

Local Scope

A variable with local scope can be accessed only within the function/block that it is created in. When a variable is created inside the function/block, the variable becomes local to it. A local variable only exists while the function is executing.

Example

```
X=50
def test ( ):
    y = 20
    print 'Value of x is ', X, ';  y is ' , y
print 'Value of x is ', X, ' y is ' , y
```

On executing the code we will get

Value of x is 50; y is 20

The next print statement will produce an error, because the variable **y** is not accessible outside the function body.

A global variable remains global, till it is not recreated inside the function/block.

Example

```
x=50
def test ( ):
    x=5
    y=2
    print 'Value of x & y inside the function are ' , x , y
print 'Value of x outside the function is ' , x
```

This code will produce following output:

Value of x & y inside the function are 5 2

Value of x outside the function is 50

If we want to refer to global variable inside the function then keyword global will be prefixed with it.

Example

```
x=50
```

```
def test ( ):
        global x
        x =2
        y = 2
        print 'Value of x & y inside the function are ' , x , y
print 'Value of x outside function is ' , x
```

This code will produce following output:

```
Value of x & y inside the function are 2  2
Value of x outside the function is 2
```

More on defining Functions

It is possible to provide parameters of function with some default value. In case the user does not want to provide values (argument) for all of them at the time of calling, we can provide default argument values.

Example

```
                                          Default value to parameter
def  greet  (message, times=1):
        print message * times
>>> greet ('Welcome')      # calling function with one argument value
>>> greet ('Hello', 2)     # calling function with both the argument values.
```

Will result in:

```
Welcome
HelloHello
```

The function **greet ()** is used to print a message (string) given number of times. If the second argument value, is not specified, then parameter **times** work with the default value provided to it. In the first call to greet ( ), only one argument value is provided, which is passed on to the first parameter from LHS and the **string is printed only once**

as the variable times take default value 1. In the second call to greet ( ), we supply both the argument values a **string** and **2**, saying that we want to print the message twice. So now, parameter **times** get the value **2** instead of default 1 and the message is printed twice.

As we have seen functions with default argument values, they can be called in with fewer arguments, then it is designed to allow.

Note:

- The default value assigned to the parameter should be a constant only.
- Only those parameters which are at the end of the list can be given default value. You cannot have a parameter on left with default argument value, without assigning default values to parameters lying on its right side.
- The default value is evaluated only once, at the point of function definition.

If there is a function with many parameters and we want to specify only some of them in function call, then value for such parameters can be provided by using their **name**, instead of the position (order)- this is called **keyword arguments**.

```
def fun(a, b=1, c=5):
        print 'a is ', a, 'b is ', b, 'c is ', c
```

The function fun can be invoked in many ways

1. >>>fun (3)

   a is 3 b is 1 c is 5

2. >>>fun (3, 7, 10)

   a is 3 b is 7 c is 10

3. >>>fun (25, c = 20)

   a is 25 b is 1 c is 20

4. >>>fun (c = 20, a = 10)

   a is 10 b is 1 c is 20

1st and 2nd call to function is based on default argument value, and the 3rd and 4th call are using **keyword arguments.**

In the first usage, value 3 is passed on to **a, b** & **c** works with default values. In second call, all the three parameters get values in function call statement. In third usage, variable **a** gets the first value 25, due to the position of the argument. And parameter **c** gets the value 20 due to naming, i.e., keyword arguments. The parameter **b** uses the default value.

In the fourth usage, we use keyword argument for all specified value, as we have specified the value for **c** before **a**; although **a** is defined before **c** in parameter list.

Note: The function named fun ( ) have three parameters out of which first one is without default value and other two have default values. So any call to the function should have at least one argument.

While using keyword arguments, following should be kept in mind:

- An argument list must have any positional arguments followed by any keywords arguments.
- Keywords in argument list should be from the list of parameters name only.
- No parameter should receive value more than once.
- Parameter names corresponding to positional arguments cannot be used as keywords in the same calls.

Following calls to fun () would be invalid

```
fun ()               # required argument missing
fun (5, a=5, 6)  # non keyword argument (6) following keyword argument
fun (6, a=5)         # duplicate value for argument a
fun (d=5)            # unknown parameter
```

Advantages of writing functions with keyword arguments are:

- Using the function is easier as we do not need to remember about the order of the arguments.
- We can specify values of only those parameters to which we want to, as - other parameters have default argument values.

In python, as function definition happens at run time, so functions can be bound to other names. This allow us to

(i) Pass function as parameter

(ii) Use/invoke function by two names

Example

```
def x ( ):
    print 20
    >>> y=x
    >>>x ( )
    >>>y ( )
    20
```

Example

```
def x ( ):
    print 20
    def test (fn):
    for I in range (4):
    fn( )
    >>> test (x)
    20
    20
```

```
20

20
```

Flow of Execution of program containing Function call

Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom. Function definition does not alter the flow of execution of program, as the statement inside the function is not executed until the function is called.

On a function call, instead of going to the next statement of program, the control jumps to the body of the function; executes all statements of the function in the order from top to bottom and then comes back to the point where it left off. This remains simple, till a function does not call another function. Simillarly, in the middle of a function, program might have to execute statements of the other function and so on.

Don't worry; Python is good at keeping track of execution, so each time a function completes, the program picks up from the place it left last, until it gets to end of program, where it terminates.

Note:

- ✡ Python does not allow you to call a function before the function is declared.
- ✡ When you write the name of a function without parenthesis, it is interpreted as the reference, when you write the function name with parenthesis, the interpreter invoke the function (object).

EXERCISE

1. The place where a variable can be used is called its

   a) area b) block

   c) function d) Scope

2. True or False

   i. Every variable has a scope associated with it.

   ii. ! (p or q) is same as !p or !q

3. What will be the output of the following? Explain:

```
def f1 ( ):
    n = 44
def f2( ):
    n=77
    print "value of n", n
    print "value of n", n
```

4. For each of the following functions. Specify the type of its **output**. You can assume each function is called with an appropriate argument, as specified by its docstrings.

   a)
```
def a (x):
'''
x: int or float.
'''
return x+1
```

   b)
```
def b (x):
'''
x: int or float.
'''
```

```
        return x+1.0
c)      def c (x, y):
            '''
            x: int or float.
            y: int or float.
            '''
            return x+y
d)      def e (x, y,z):
            '''
            x: can be of any type.
            y: can be of any type.
            z: can be of any type
            '''
            return x >= y and x <= z
e)      def d (x,y):
            '''
            x: can be of any type.
            y: can be of any type.
            '''
            return x > y
```

5. Below is a transcript of a session with the Python shell. Assume the functions in previous question (Q 4) have been defined. Provide the type and value of the expressions being evaluated.

   i) a (6)　　ii) a (-5. 3)

   iii) a (a(a(6)))　　iv) c (a(1), b(1))

   v) d ('apple', 11.1)

6. Define a function **get Bigger Number (x,y)** to take in two numbers and return the bigger of them.

7. What is the difference between methods, functions & user defined functions.

8. Open help for math module

   i. How many functions are there in the module?

   ii. Describe how square root of a value may be calculated without using a math module

   iii. What are the two data constants available in math module.

9. Generate a random number *n* such that

   i. $0 \leq n < 6$

   ii. $2 \leq n < 37$ and *n* is even

LAB EXERCISE

1. Write a program to ask for following as input

   Enter your first name: Rahul

   Enter your last name: Kumar

   Enter your date of birth

   Month? March

   Day? 10

   Year? 1992

   And display following on screen

   Rahul Kumar was born on March 10, 1992.

2. Consider the following function definition:

```
def intDiv (x, a):
    """
    x: a non-negative integer argument
```

```
a: a positive integer argument

returns: integer, the integer division of x divided by a.

"""

while x>=a:

count +=1

x = x-a

return count
```

when we call

print intDiv (5, 3)

We get an error message. Modify the code so that error does not occur.

3. Write a script that asks a user for a number. Then adds 3 to that number, and then multiplies the result by 2, subtracts twice the original number, then prints the result.

4. In analogy to the example, write a script that asks users for the temperature in F and prints the temperature in C. (Conversion: Celsius = (F - 32) * 5/9).

5. Write a Python function, odd, that takes in one number and returns True when the number is odd and False otherwise. You should use the % (mod) operator, not **if**.

6. Define a function 'SubtractNumber(x,y)' which takes in two numbers and returns the difference of the two.

7. Write a Python function, fourthPower( ), that takes in one number and returns that value raised to the fourth power.

8. Write a program that takes a number and calculate and display the log, square, sin and cosine of it.

9. a) Write a program, to display a tic-tac-toe board on screen, using print statement.

   b) Write a program to display a tic-tac-toe board on screen using variables, so that you do not need to write many print statements?

10. Write a function roll_D ( ), that takes 2 parameters- the no. of sides (with default value 6) of a dice, and the number of dice to roll-and generate random roll values for each dice rolled. Print out each roll and then return one string "That's all".

Example roll_D (6, 3)

```
4

1

6

That's all
```