CHAPTER

9

FUNCTIONS

A function is similar to a program that consists of a group of statements that are intended to perform a specific task. The main purpose of a function is to perform a specific task or work. Thus when there are several tasks to be performed, the programmer will write several functions. There are several 'built-in' functions in Python to perform various tasks. For example, to display output, Python has print() function. Similarly, to calculate square root value, there is sqrt() function and to calculate power value, there is power() function. Similar to these functions, a programmer can also create his own functions which are called 'user-defined' functions. The following are the advantages of functions:

- Functions are important in programming because they are used to process data, make calculations or perform any task which is required in the software development.
- Once a function is written, it can be reused as and when required. So functions are also called reusable code. Because of this reusability, the programmer can avoid code redundancy. It means it is possible to avoid writing the same code again and again.
- Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules. To represent each module, the programmer will develop a separate function. Then these functions are called from a main program to accomplish the complete task. Modular programming makes programming easy.
- Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software. Similarly, when a particular feature is no more needed by the user, the corresponding function can be deleted or put into comments.

- When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software. Thus code debugging will become easy.
- The use of functions in a program will reduce the length of the program.

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, please remember that a function and a method are 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. Consider the syntax of function, as shown in Figure 9.1:

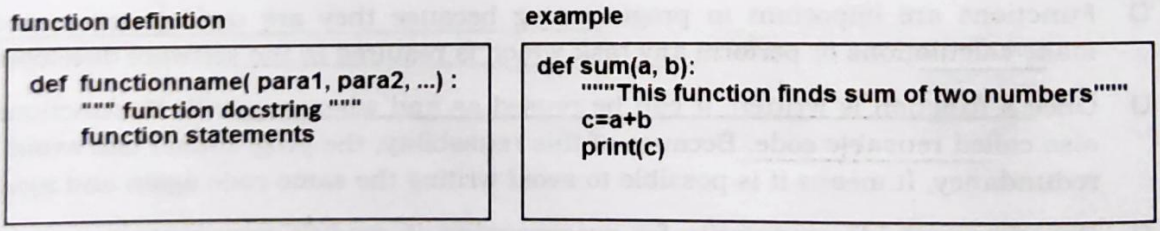

Figure 9.1: Understanding the Function Definition

For example, we can write a function to add two values as:

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

Here, 'def' represents the starting of function definition. 'sum' is the name of the function. After this name, parentheses ( ) are compulsory as they denote that it is a function and not a variable or something else. In the parentheses, we wrote two variables 'a' and 'b'. These variables are called 'parameters'. A parameter is a variable that receives data from outside into a function. So, this function can receive two values from outside and those values are stored in the variables 'a' and 'b'.

Please understand that Python treats 1 as True and 0 as False. Hence the above statement reads as:

```
if True:
    print(i)
```

Hence, the print() function is executed and 'i' value is displayed.

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 'c' will be stored into 'x' and the result of subtraction which is in 'd' will be stored into 'y'. This is shown in Program 7.

Program

Program 7: A Python program to understand how a function returns two values.

```
# a function that returns two results
def sum_sub(a, b):
    """ this function returns results of
        addition and subtraction of a, b """
    c = a + b
    d = a - b
    return c, d

# get the results from the sum_sub() function
x, y = sum_sub(10, 5)
```

```
# display the results
print("Result of addition: ", x)
print("Result of subtraction: ", y)
```

Output:

```
C:\>python fun.py
Result of addition:  15
Result of subtraction:  5
```

When several results are returned, we can retrieve them from the tuple using a for loop or while loop. This is shown in Program 8. This is an extension to Program 7.

Program

Program 8: A function that returns the results of addition, subtraction, multiplication and division.

```
# a function that returns multiple results
def sum_sub_mul_div(a, b):
    """ this function returns results of addition,
        subtraction, multiplication and division of a, b """
    c = a + b
    d = a - b
    e = a * b
    f = a / b
    return c, d, e, f

# get results from sum_sub_mul_div() function and store into t
t = sum_sub_mul_div(10, 5)

# display the results using for loop
print('The results are: ')
for i in t:
    print(i, end=', ')
```

Output:

```
C:\>python fun.py
The results are:
15, 5, 50, 2.0,
```

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 parameter to another function.
- It is possible that a function can return another function.

To understand these points, we will take a few simple programs. In Program 9, 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'.

Program

Program 9: A Python program to see how to assign a function to a variable.

```
# assign a function to a variable
def display(str):
    return 'Hai '+str

# assign function to variable x
x = display("Krishna")
print(x)
```

Output:

```
C:\>python fun.py
Hai Krishna
```

In Program 10, we write message() function inside display() function. We call message() function from display() and return the result.

Program

Program 10: A Python program to know how to define a function inside another function.

```
# define a function inside another function
def display(str):
    def message():
        return 'How are U?'

    result = message()+str
    return result

# call display() function
print(display("Krishna"))
```

Output:

```
C:\>python fun.py
How are U? Krishna
```

In Program 11, we are trying to pass message() function as a parameter or argument to display() function. Since display() function has to receive another function as its parameter, it should be defined with another function 'fun' as parameter as:

```
def display(fun):
```

Program

Program 11: A Python program to know how to pass a function as parameter to another function.

```
# functions can be passed as parameters to other functions
def display(fun):
    return 'Hai '+ fun

def message():
    return 'How are U? '

# call display() function and pass message() function
print(display(message()))
```

Output:

```
C:\>python fun.py
Hai How are U?
```

In Program 12, we are writing message() function inside display() function. Inside the display() function, when we write:

```
return message
```

This will return the message() function out of display() function. The returned message() function can be referenced with a new name 'fun'.

Program

Program 12: A Python program to know how a function can return another function.

```
# functions can return other functions
def display():
    def message():
        return 'How are U?'

    return message

# call display() function and it returns message() function
# in the following code, fun refers to the name: message.
fun = display()
print(fun())
```

Output:

```
C:\>python fun.py
How are U?
```

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):  # a, b are formal arguments
    c = a+b
    print(c)

# call the function
x=10; y=15
sum(x, y)   # x, y are actual arguments
```

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.

Program

✓**Program 16:** A Python program to understand the positional arguments of a function.

```
# positional arguments demo
def attach(s1, s2):
    """ to join s1 and s2 and display total string """
    s3 = s1+s2
    print('Total string: '+s3)

# call attach() and pass 2 strings
attach('New', 'York')  # positional arguments
```

Output:

```
C:\>python fun.py
Total string: NewYork
```

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 mention 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. Program 17 demonstrates how to use keyword arguments while calling grocery() function.

Program

Program 17: A Python program to understand the keyword arguments of a function.

```
# key word arguments demo
def grocery(item, price):
    """ to display the given arguments """
    print('Item = %s' % item)
    print('Price = %.2f' % price)

# call grocery() and pass 2 arguments
grocery(item='Sugar', price=50.75)   # keyword arguments
grocery(price=88.00, item='Oil')  # keyword arguments
```

Output:

```
C:\>python fun.py
Item = Sugar
Price = 50.75
Item = Oil
Price = 88.00
```

Default Arguments

We can mention some default value 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. Program 18 will clarify this.

Program

Program 18: A Python program to understand the use of default arguments in a function.

```
# default arguments demo
def grocery(item, price=40.00):
    """ to display the given arguments """
    print('Item = %s' % item)
```

```
    print('Price = %.2f' % price)

# call grocery() and pass arguments
grocery(item='Sugar', price=50.75)   # pass 2 arguments
grocery(item='Sugar')  # default value for price is used.
```

Output:

```
C:\>python fun.py
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 two numbers, he can write:

```
add(a, b)
```

But, the user who is using this function may want to use this function to find 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 error will be displayed. If the programmer wants to develop a function that can accept 'n' arguments, that is also possible in Python. For this 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 length 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 argument. We can pass 1 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 Program 19, we are showing how to use variable length argument.

Program

Program 19: A Python program to show variable length argument and its use.

```
# variable length argument demo
def add(farg, *args):   # *args can take 0 or more values
    """ to add given numbers """
    print('Formal argument= ', farg)

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

```
# call add() and pass arguments
add(5, 10)
add(5, 10, 20, 30)
```

Output:

```
C:\>python fun.py
Formal argument=  5
Sum of all numbers=  15
Formal argument=  5
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 is 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 'a' is removed from memory and it is not available. Consider the following code:

```
# local variable in a function
def myfunction():
    a=1   # this is local var
    a+=1   # increment it
    print(a)   # displays 2

myfunction()
print(a)   # error, not available
```

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 global variable. Such variables are available to all the functions which are written after it. Consider the following code:

```
# global variable example
a=1   # this is global var
def myfunction():
    b=2   # this is local var
    print('a= ', a)   # display globalvar
    print('b= ', b)   # display localvar

myfunction()
print(a)   # available
print(b)   # error, not available
```

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 that 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 Program 21.

Program

Program 21: A Python program to understand global and local variables.

```
# same name for global and local variables
a=1   # this is global var
def myfunction():
    a=2   # this is local var
    print('a= ', a)   # display local var
```

```
myfunction()
print('a= ', a)   #display global var
```

Output:

```
C:\>python fun.py
a= 2
a= 1
```

When the programmer wants to use the global variable inside a function, he can use the keyword 'global' before the variable in the beginning of the function body as:

```
global a
```

In this way, the global variable is made available to the function and the programmer can work with it as he wishes. In the following program, we are showing how to work with a global variable inside a function.

Program

Program 22: A Python program to access global variable inside a function and modify it.

```
# accessing the global variable inside a function
a=1   # this is global var
def myfunction():
    global a   # this is global var
    print('global a= ', a)   # display global var
    a=2   # modify global var value
    print('modifed a= ', a)   # display new value

myfunction()
print('global a= ', a)   # display modified value
```

Output:

```
C:\>python fun.py
global a= 1
modified a= 2
global a= 2
```

When the global variable name and local variable names are same, the programmer will face difficulty to differentiate between them inside a function. For example there is a global variable 'a' with some value declared above the function. The programmer is writing a local variable with the same name 'a' with some other value inside the function. Consider the following code:

```
a=1   # this is global var
def myfunction():
    a = 2   # a is local var
```

Now, if the programmer wants to work with global variable, how is it possible? If he uses 'global' keyword, then he can access only global variable and the local variable is no more available.

The concept of recursion helps the programmers to solve complex programs in less number of steps. Imagine if we attempt to write the Towers of Hanoi program without using recursion, it would have been a highly difficult task.

Anonymous Functions or Lambdas

A function without a name is called 'anonymous function'. So far, the functions we wrote were defined using the keyword 'def'. But anonymous functions are not defined using 'def'. They are defined using the keyword *lambda*and hence they are also called 'Lambda functions'. Let's take a normal function that returns square of a given value.

```
def square(x):
    return x*x
```

The same function can be written as anonymous function as:

```
lambda x: x*x
```

Observe the keyword 'lambda'. This represents that an anonymous function is being created. After that, we have written an argument of the function, i.e. 'x'. Then colon (:) represents the beginning of the function that contains an expression x * x. Please observe that we did not use any name for the function here. So, the format of lambda functions is:

```
lambda argument_list : expression
```

Normally, if a function returns some value, we assign that value to a variable as:

```
y = square(5)
```

But, lambda functions return a function and hence they should be assigned to a function as:

```
f = lambda x: x*x
```

Here, 'f' is the function name to which the lambda expression is assigned. Now, if we call the function f() as:

```
value = f(5)
```

Now, 'value' contains the square value of 5, i.e. 25. This is shown in Program 28.

Program

Program 28: A Python program to create a lambda function that returns a square value of a given number.

```
# a lambda function to calculate square value
f = lambda x: x*x   # write lambda function
value = f(5)   # call lambda function
print('Square of 5 = ', value)   # display result
```

Output:

```
C:\>python fun.py
Square of 5 = 25
```

Lambda functions contain only one expression and they return the result implicitly. Hence we should not write any 'return' statement in the lambda functions. Here is a lambda function that calculates sum of two numbers.

Program

Program 29: A lambda function to calculate the sum of two numbers.

```
# a lambda function to calculate sum of two numbers
f = lambda x, y: x+y  # write lambda function
result = f(1.55, 10)   # call lambda function
print('Sum = ', result)   # display result
```

Output:

```
C:\>python fun.py
Sum =  11.55
```

The following is a program that contains a lambda function to find the bigger number in two given numbers.

Program

Program 30: A lambda function to find the bigger number in two given numbers.

```
# a lambda function that returns bigger number
max = lambda x, y: x if x>y else y  # write lambda function
a, b = [int(n) for n in input("Enter two numbers: ").split(',')]
print('Bigger number = ', max(a,b))   # call lambda function
```

Output:

```
C:\>python fun.py
Enter two numbers: 10, 25
Bigger number =  25
```

Because a lambda functions is always represented by a function, we can pass a lambda function to another function. It means we are passing a function (i.e. lambda) to another function. This makes processing the data very easy. For example, lambda functions are generally used with functions like filter(), map() or reduce().

Creating our Own Modules in Python

A module represents a group of classes, methods, functions and variables. While we are developing software, there may be several classes, methods and functions. We should first group them depending on their relationship into various modules and later use these

modules in the other programs. It means, when a module is developed, it can be reused in any program that needs that module.

In Python, we have several built-in modules like sys, io, time etc. Just like these modules, we can also create our own modules and use them whenever we need them. Once a module is created, any programmer in the project team can use that module. Hence, modules will make software development easy and faster.

Now, we will create our own module by the name 'employee' and store the functions da(), hra(), pf() and itax() in that module. Please remember we can also store classes and variables etc. in the same manner in any module. So, please type the following functions and save the file as 'employee.py'.

```
# save this code as employee.py
# to calculate dearness allowance
def da(basic):
    """ da is 80% of basic salary """
    da = basic*80/100
    return da

# to calculate house rent allowance
def hra(basic):
    """ hra is 15% of basic salary """
    hra = basic*15/100
    return hra

# to calculate provident fund amount
def pf(basic):
    """ pf is 12% of basic salary """
    pf = basic*12/100
    return pf

# to calculate income tax payable by employee
def itax(gross):
    """ tax is calculated at 10% on gross """
    tax = gross*0.1
    return tax
```

Now, we have our own module by the name 'employee.py'. This module contains 4 functions which can be used in any program by importing this module. To import the module, we can write:

```
import employee
```

In this case, we have to refer to the functions by adding the module name as: employee.da(), employee.hra(), employee.pf(), employee.itax(). This is a bit cumbersome and hence we can use another type of import statement as:

```
from employee import *
```

In this case, all the functions of the employee module is referenced and hence, we can refer to them without using the module name, simply as: da(), hra(), pf() and itax(). Now see the following program, where we are using the 'employee' module and calculating the gross and net salaries of an employee.

Program

Program 44: A Python program that uses the functions of employee module and calculates the gross and net salaries of an employee.

```
# using employee module to calculate gross and net salaries of an employee
from employee import *

# calculate gross salary of employee by taking basic
basic = float(input('Enter basic salary: '))

# calculate gross salary
gross = basic+da(basic)+hra(basic)
print('Your gross salary: {:10.2f}'. format(gross))

# calculate net salary
net = gross - pf(basic)-itax(gross)
print('Your net salary: {:10.2f}'. format(net))
```

Output:

```
C:\>python fun.py
Enter basic salary: 15000
Your gross salary:     29250.00
Your net salary:     24525.00
```