3

Working with Functions

In This Chapter

3.1 Introduction
3.2 Understanding Functions
3.3 Defining Functions in Python
3.4 Flow of Execution in a Function Call
3.5 Passing Parameters
3.6 Returning Values From Functions
3.7 Composition
3.8 Scope of Variables

3.1 INTRODUCTION

Large programs are generally avoided because it is difficult to manage a single list of instructions. Thus, a large program is broken down into smaller units known as functions. A function is a named unit of a group of program statements. This unit can be invoked from other parts of the program.

The most important reason to use functions is to make program handling easier as only a small part of the program is dealt with at a time, thereby avoiding ambiguity. Another reason to use functions is to reduce program size. Functions make a program more readable and understandable to a programmer thereby making program management much easier.

In this chapter, we shall talk about functions, especially, how a function works ; how you can create your own functions in Python ; and how you can use the functions created by you.

FUNCTION

A *Function* is a subprogram that acts on data and often returns a value.

Scanned with CamScanner

3.2 UNDERSTANDING FUNCTIONS

In order to understand what a function is, in terms of a programming language, read the following lines carefully.

You have worked with polynomials in Mathematics. Say we have following polynomial :

$2x^2$

For $x = 1$, it will give result as $2 \times 1^2 = 2$

For $x = 2$, it will give result as $2 \times 2^2 = 8$

For $x = 3$, it will give result as $2 \times 3^2 = 18$

and so on.

Now, if we represent above polynomial as somewhat like

$f(x) = 2x^2$

Then we can say (from above calculations) that

$$f(1) = 2 \quad ...(1)$$

$$f(2) = 8 \quad ...(2)$$

$$f(3) = 18 \quad ...(3)$$

The notation $f(x) = 2x^2$ can be termed as a **function**, where for function namely f, x is its argument *i.e.*, value given to it, and $2x^2$ is its functionality, *i.e.*, the functioning it performs. For different values of argument x, function $f(x)$ will return different results (refer to equations (1), (2) and (3) given above).

On the similar lines, programming languages also support functions. You can create functions in a program, that :

- can have arguments (*values given to it*), if needed
- can perform certain functionality (*some set of statements*)
- can return a result

For instance, above mentioned mathematical function $f(x)$ can be written in Python like this :

```
def calcSomething ( x ) :
    r = 2 * x ** 2
    return r
```

where

- **def** means a function definition is starting
- identifier following 'def' is the name of the function, *i.e.*, here the function name is ***calcSomething***
- the variables/identifiers inside the parentheses are the *arguments* or *parameters* (values given to function), *i.e.*, here x is the argument to function ***calcSomething***.
- there is a colon at the end of *def* line, meaning it requires a block

Scanned with CamScanner

- the statements indented below the function, (*i.e.*, block below *def* line) define the functionality (working) of the function. This block is also called ***body-of-the-function***. Here, there are ***two statements in the body of function calcSomething***.
- The **return** statement returns the computed result.

The non-indented statements that are below the function definition are not part of the function **calcSomething**'s definition. For instance, consider the example-function given in Fig. 3.1 below :

To see
Function anatomy
in action

Scan
QR Code

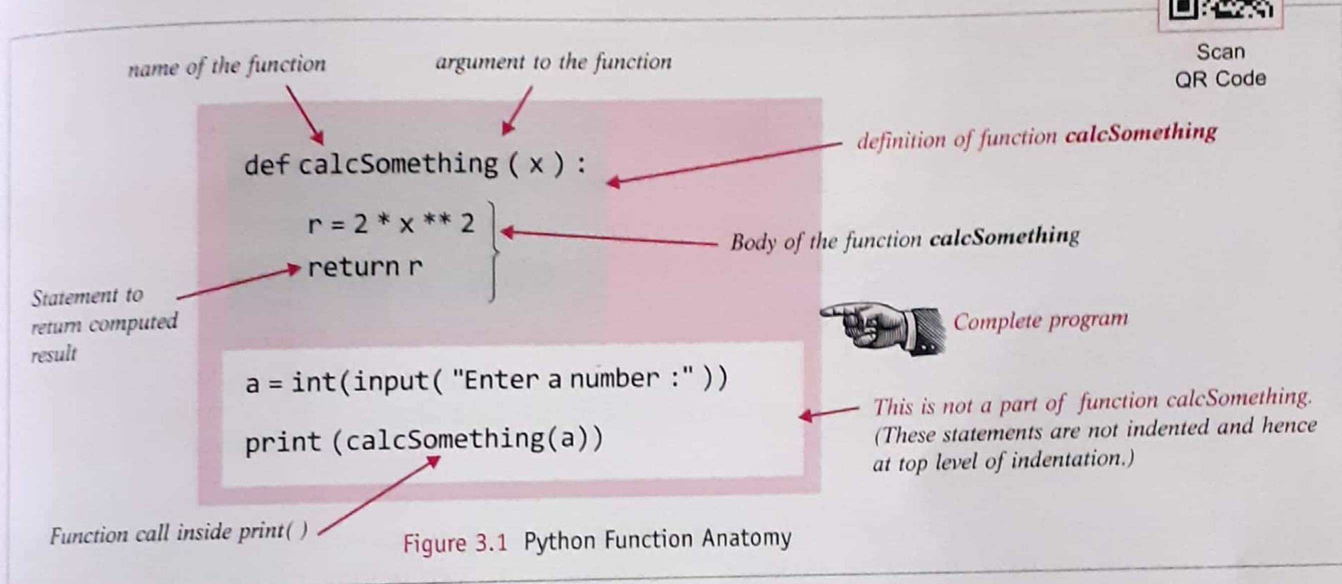


Figure 3.1 Python Function Anatomy

3.2.1 Calling/Invoking/Using a Function

To use a function that has been defined earlier, you need to write a *function call* statement in Python. A *function call* statement takes the following form :

```
<function-name>(<value-to-be-passed-to-argument>)
```

For example, if we want to call the function **calcSomething( )** defined above, our function call statement will be like :

```
calcSomething(5)            # value 5 is being sent as argument
```

Another function call for the same function, could be like :

```
a = 7
calcSomething(a)            # this time variable a is being sent as argument
```

Carefully notice that number of values being passed is same as number of parameters.

Also notice, in Fig. 3.1, the last line of the program uses a function call statement. (*print* ( ) is using the function call statement.)

To see
Part of a function
in action

Scan
QR Code

Consider one more function definition given below :

```
def cube(x) :
    res = x ** 3            # cube of value in x
    return res              # return the computed value
```

Scanned with CamScanner

As you can make out that the above function's name is **cube( )** and it takes one argument. Now its function call statement(s) would be similar to the ones shown below :

(*i*) Passing literal as argument in function call

```
cube(4)          # it would pass value as 4 to argument x
```

(*ii*) Passing variable as argument in function call

```
num = 10
cube(num)        # it would pass value as variable num to argument x
```

(*iii*) taking input and passing the input as argument in function call

```
mynum = int (input ( "Enter a number :" ))
cube(mynum)      # it would pass value as variable mynum to argument x
```

(*iv*) using function call inside another statement

```
print(cube(3))   # cube(3) will first get the computed result
                 # which will be then printed
```

(*v*) using function call inside expression

```
doubleOfCube = 2 * cube(6)
                 # function call's result will be multiplied with 2
```

NOTE

The syntax of the function call is very similar to that of the declaration, except that the key word ***def*** and colon (:) are missing.

3.2.2 Python Function Types

Python comes preloaded with many *function-definitions* that you can use as per your needs. You can even create new functions. Broadly, Python functions can belong to one of the following *three* categories :

1. **Built-in functions** These are pre-defined functions and are always available for use. You have used some of them – *len( ), type( ), int( ), input( )* etc.

2. **Functions defined in modules** These functions are pre-defined in particular modules and can only be used when the corresponding module is *imported*. *For example*, if you want to use pre-defined functions inside a module, say **sin( )**, you need to first *import* the module *math* (that contains definition of **sin( )**) in your program.

3. **User defined functions** These are defined by the programmer. As programmers you can create your own functions.

In this chapter, you will learn to write your own Python functions and use them in your programs.

STRUCTURE OF FUNCTIONS

PriP — Progress In Python 3.1

This PriP session is aimed at making anatomy of Python functions clear to you. You'll be required to practice about structure of Functions.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.1 under Chapter 3 after practically doing it on the computer.

Scanned with CamScanner

3.3 DEFINING FUNCTIONS IN PYTHON

As you know that we write programs to do certain things. Functions can be thought of as key-doers within a program. A function once defined can be invoked as many times as needed by using its name, without having to rewrite its code.

In the following lines, we are about to give the general form *i.e.*, syntax of writing function code in Python. Before we do that, just remember these things.

In a syntax language :

- item(s) inside angle brackets <> has to be provided by the programmer.
- item(s) inside square brackets [ ] is optional, *i.e.*, can be omitted.
- items/words/punctuators outside <> and [ ] have to be written as specified.

A function in Python is defined as per following general format :

```
def <function name> ( [parameters] ) :
    [ """<function's docstring>""" ]
    <statement>
    [<statement>]
    :
```

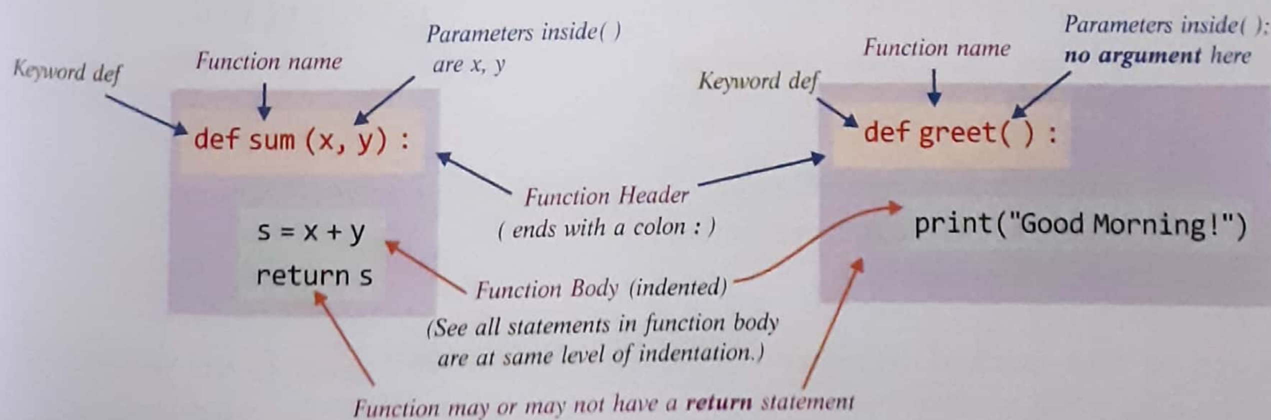
For example, consider some function definitions given below:

```
def sum (x, y) :
    s = x + y
    return s
```

Or

```
def greet( ):
    print("Good Morning!")
```

Though you know about various elements in a function-definition, still let us talk about it again. Let us dissect these functions' definitions to know about various components.



Let us define these terms formally :

Function Header The first line of function definition that begins with keyword *def* and ends with a colon (:), specifies the ***name of the function*** and its ***parameters***

Scanned with CamScanner

Parameters	Variables that are listed within the parentheses of a function header
Function Body	The block of statements / indented-statements beneath function header that defines the action performed by the function. The function body may or may not return any value. A function returns a value through a *return* statement, *e.g.*, above given ***sum( )*** is returning a value stored in variables, but function ***greet( )*** is not returning a value. A function not returning any value can still have a *return* statement without any expression or value. Examples below will make it clearer.
Indentation	The blank space in the beginning of a statement (convention is four spaces) within a block. All statements within same block have same indentation.

Let us now have a look at some more function definitions.

```
# Sample Code 1

def sumOf3Multiples1( n ) :
    s = n * 1 + n * 2 + n * 3
    return s
```

Both these functions are doing the same thing BUT first one is ***returning*** *the computed value using return statement and second function is* ***printing*** *the computed value using print( ) statement*

```
# Sample Code 2
def sumOf3Multiples2( n ) :
    s = n * 1 + n * 2 + n * 3
    print(s)
```

Consider some more function definitions:

```
# Sample Code 3
def areaOfSquare ( a ) :
    return a * a
```

```
# Sample Code 4
def areaOfRectangle ( a, b) :
    return a * b
```

```
# Sample Code 5
def perimeterCircle( r ) :
    return (2 * 3.1459 * r)
```

```
# Sample Code 6
def perimeterRectangle( l, b) :
    return 2 * ( l + b )
```

```
# Sample Code 7
def Quote( ) :
    print("\t Quote of the Day")
    print("Act Without Expectation!!")
    print("\t -Lao Tzu")
```

For all these function definitions, try identifying their parts. (Not as an exercise, just do it casually, while reading them.)

A function definition defines a user-defined object function. The function definition does not execute the function body; this gets executed only when the function is called or invoked. In the following lines, we are discussing how to invoke functions, but before that it would be useful to know the basic structure of a Python program.

Scanned with CamScanner

3.3.1 Structure of a Python Program

In a Python program, generally all function definitions are given at the top followed by statements which are not part of any functions. These statements are not indented at all. These are often called from the top-level statements (the ones with no indentation). The Python interpreter starts the execution of a program/script from the top-level statements. The top level statements are part of the main program. Internally Python gives a special name to top-level statements as ***__main__*** .

NOTE

By default, Python names the segment with top-level statements (main program) as ***__main__*** .

The structure of a Python program is generally like the one shown below :

```
def function1( ) :
    :
def function2( ) :
    :
def function3( ) :
    :
    :
# top-level statements here
statement1
statement2
    :
```

Python names the segment with top-level statements (no identation) as __main__. Python begins execution of a program from the top-level statements i.e., from __main__ .

Python stores this name in a built-in variable called **__name__** (*i.e.*, you need not declare this variable ; you can directly use it). You can see it yourself. In the ***__main__*** segment of your program if you give a statement like :

```
print(__name__)
```

Python will show you this name. For example, run the following code and see it yourself.

```
def greet( ) :
    print("Hi there!")

print("At the top-most level right now")
print("Inside", __name__)
```

The top-level statements, i.e., the __main__ segment of this Python program. Python will start execution of this program from the segment.

Upon executing above program, Python will display :

```
At the top-most level right now
Inside __main__
```

Notice word ' ***__main__*** *' in the output by Python interpreter. This is the result of statement :*
print(..., __name__)

Parameters	Variables that are listed within the parentheses of a function header
Function Body	The block of statements / indented-statements beneath function header that defines the action performed by the function. The function body may or may not return any value. A function returns a value through a *return* statement, *e.g.*, above given ***sum( )*** is returning a value stored in variables, but function ***greet( )*** is not returning a value. A function not returning any value can still have a *return* statement without any expression or value. Examples below will make it clearer.
Indentation	The blank space in the beginning of a statement (convention is four spaces) within a block. All statements within same block have same indentation.

Let us now have a look at some more function definitions.

```
# Sample Code 1
def sumOf3Multiples1( n ) :
    s = n * 1 + n * 2 + n * 3
    return s
```

Both these functions are doing the same thing BUT first one is ***returning*** *the computed value using return statement and*

second function is ***printing*** *the computed value using print( ) statement*

```
# Sample Code 2
def sumOf3Multiples2( n ) :
    s = n * 1 + n * 2 + n * 3
    print(s)
```

Consider some more function definitions:

```
# Sample Code 3
def areaOfSquare ( a ) :
    return a * a
```

```
# Sample Code 4
def areaOfRectangle ( a, b) :
    return a * b
```

```
# Sample Code 5
def perimeterCircle( r ) :
    return (2 * 3.1459 * r)
```

```
# Sample Code 6
def perimeterRectangle( l, b) :
    return 2 * ( l + b )
```

```
# Sample Code 7
def Quote( ) :
    print("\t Quote of the Day")
    print("Act Without Expectation!!")
    print("\t -Lao Tzu")
```

For all these function definitions, try identifying their parts. (Not as an exercise, just do it casually, while reading them.)

A function definition defines a user-defined object function. The function definition does not execute the function body; this gets executed only when the function is called or invoked. In the following lines, we are discussing how to invoke functions, but before that it would be useful to know the basic structure of a Python program.

Scanned with CamScanner

3.4 FLOW OF EXECUTION IN A FUNCTION CALL

Let us now talk about how the control flows (*i.e.*, the flow of execution of statements) in case of a function call. You already know that a function is called (or invoked, or executed) by providing the function name, followed by the values being sent enclosed in parentheses. For instance, to invoke a function whose header looks like :

```
def sum (x, y) :
```

the *function call statement* may look like as shown below :

```
sum (a, b)
```

where *a*, *b* are the values being passed to the function *sum*( ).

Let us now see what happens when Python interpreter encounters a function call statement.

The *Flow of Execution* refers to the order in which statements are executed during a program run.

Recall that a block is a piece of Python program text that is executed as a unit (denoted by line indentation). A **function body** is also a block. In Python, a block is executed in an **execution frame**.

An execution frame contains :

- some internal information (used for debugging)
- name of the function
- values passed to function
- variables created within function
- information about the next instruction to be executed.

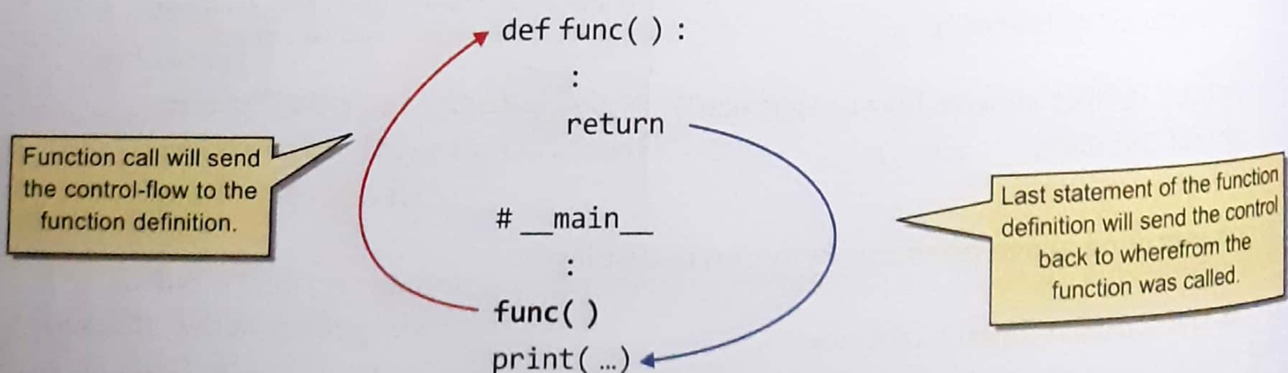
FLOW OF EXECUTION

The ***Flow of Execution*** refers to the order in which statements are executed during a program run.

NOTE

The Flow of Execution refers to the order in which statements are executed during a program run.

Whenever a function call statement is encountered, an *execution frame* for the called function is created and the control (program control) is transferred to it. Within the function's execution frame, the statements in the function-body are executed, and with the *return statement* or the last statement of function body, the control returns to the statement wherefrom the function was called, *i.e.*, as :



Let us now see how all this is done with the help of an example. Consider the following program 3.1 code.

Scanned with CamScanner

Program 3.1 Program to add two numbers through a function

```
# program add.py to add two numbers through a function
def calcSum (x, y) :
    s = x + y                # statement 1
    return s                 # statement 2

num1 = float(input( "Enter first number :" ) )       # 1 (statement 1)
num2 = float(input( "Enter second number :" ) )      # 2 (statement 2)
sum = calcSum(num1, num2)                            # 3 (statement 3)
print("Sum of two given numbers is", sum)            # 4 (statement 4)
```

To see Working of a function in action

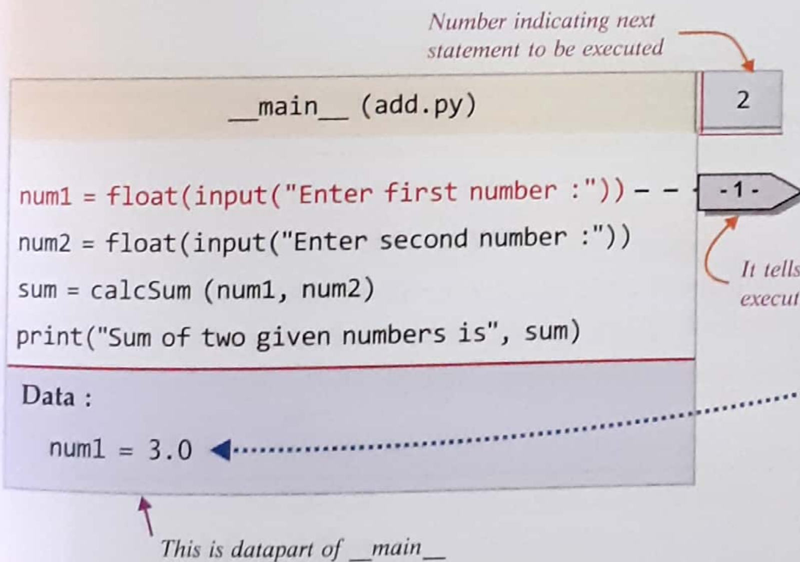
Scan QR Code

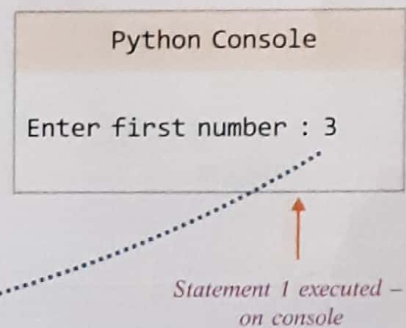
Program execution begins with first statement of *__main__* segment. (def statements are also read but ignored until called. It will become clear to you in a few moments. Just read on.)

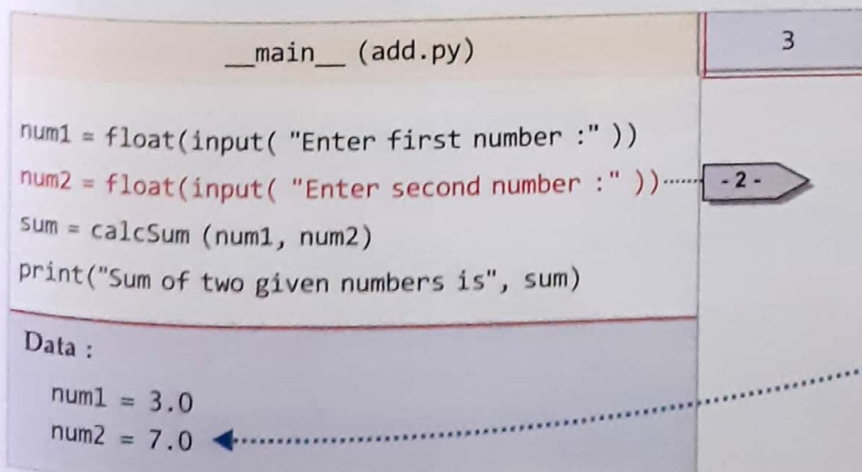
(Please note that in the following lines, we have put up some execution frames for understanding purposes only; these are not based on any standard diagram.)

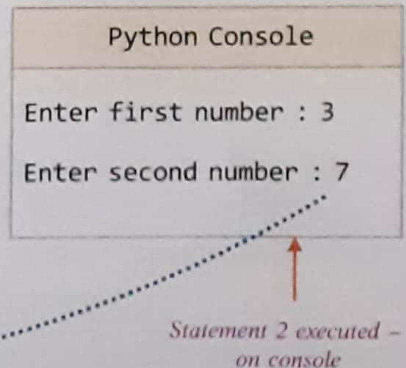
NOTE

Program execution begins with first statement of *__main__* segment.









Scanned with CamScanner

3.5 PASSING PARAMETERS

Uptill now you learnt that a function call must provide all the values as required by function definition. For instance, if a function header has *three* parameters named in its header then the function call should also pass *three* values. Other than this, Python also provides some other ways of sending and matching arguments and parameters.

Python supports *three* types of formal arguments/parameters :

1. Positional arguments (*Required arguments*)
2. Default arguments
3. Keyword (or *named*) arguments

To see
Function Parameters
in action

Scan
QR Code

Let us talk about these, one by one.

3.5.1 Positional/Required Arguments

Till now you have seen that when you create a function call statement for a given function definition, you need to match the number of arguments with number of parameters required. *For example,* if a function definition header is like :

```
def check (a, b, c) :
      :
```

then possible function calls for this can be :

```
check ( x, y, z)            # 3 values (all variables) passed
check (2, x, y)             # 3 values (literal + variables) passed
check (2, 5, 7)             # 3 values (all literals) passed
     :
```

See, in all the above function calls, the number of passed values (arguments) has matched with the number of received values (parameters). Also, the values are given (or matched) position-wise or order-wise, *i.e.,* the first parameter receives the value of first argument, second parameter, the value of second argument and so on *e.g.,*

In function call 1 above :	In function call 2 above :	In function call 3 above :
❖ *a* will get value of *x*	❖ *a* gets value of 2 ;	❖ *a* gets value 2 ;
❖ *b* will get value of *y*	❖ *b* gets value of *x* ;	❖ *b* gets value 5 ;
❖ *c* will get value of *z*	❖ *c* gets value of *y*	❖ *c* gets value 7

Thus, through such function calls,

- ❖ the arguments must be provided for all parameters (*Required*)
- ❖ the values of arguments are matched with parameters, position (order) wise (*Positional*)

This way of parameter and argument specification is called ***Positional arguments*** or ***Required arguments*** or ***Mandatory arguments*** as no value can be skipped from the function call or you cannot change the order *e.g.,* you cannot assign value of first argument to third parameter.

NOTE

When the function call statement must match the number and order of arguments as defined in the function definition, this is called the **positional argument matching**

Scanned with CamScanner

3.5.2 Default Arguments

What if we already know the value for a certain parameter, *e.g.*, in an interest calculating function, we know that mostly the rate of interest is 10%, then there should be a provision to define this value as the default value.

Python allows us to assign default value(s) to a function's parameter(s) which is useful in case a matching argument is not passed in the function call statement. The **default values are specified in the function header of function definition**. Following is an example of function header with default values :

```
def interest (principal, time, rate = 0.10) :
```

*This is default value for parameter **rate**. If in a function call, the value for **rate** is not provided, Python will fill the missing value (**for rate only**) with this value.*

The *default value* is specified in a manner syntactically similar to a variable initialization. The above function declaration provides a default value of 0.10 to the parameter ***rate***.

> **NOTE**
>
> Non-default arguments cannot follow default argument.

Now, if any function call appears as follows :

```
si_int = interest (5400, 2)          # third argument missing
```

then the value **5400** is passed to the parameter ***principal***, the value **2** is passed to the second parameter ***time*** and since the third argument ***rate*** is missing, its default value **0.10** is used for ***rate***. But if a function call provides all *three* arguments as shown below :

```
si_int = interest (6100, 3, 0.15)    # no argument missing
```

then the parameter ***principal*** gets value **6100**, ***time*** gets **3** and the parameter ***rate*** gets value ***0.15***.

> **DEFAULT PARAMETER**
>
> A parameter having default value in the function header is known as a default parameter.

That means the default values (values assigned in function header) are considered only if no value is provided for that parameter in the function call statement.

One very important thing you must know about default parameters is :

In a function header, any parameter cannot have a default value unless all parameters appearing on its right have their default values.

For instance, in the above mentioned declaration of function ***interest( )***, the parameter ***principal*** cannot have its default value unless the parameters on its right, ***time*** and ***rate*** also have their default values. Similarly, the parameter ***time*** cannot have its default value unless the parameter on its right, *i.e.*, ***rate*** has its default value. There is no such condition for ***rate*** as no parameter appears on its right.

> **NOTE**
>
> A parameter having a default value in function header becomes optional in function call. Function call may or may not have value for it.

Thus, *required parameters* should be before *default parameters*.

Scanned with CamScanner

Following are examples of function headers with default values :

```
def interest (prin, time, rate = 0.10) :              # legal
def interest (prin, time = 2, rate) :                 # illegal (default parameter
                                                      # before required parameter)
def interest (prin = 2000, time = 2, rate) :          # illegal
                                                      # (same reason as above)
def interest (prin, time = 2, rate = 0.10) :          # legal
def interest (prin = 200, time = 2, rate = 0.10) :    # legal
```

Default arguments are useful in situations where some *parameters* always have the same value. Also they provide greater flexibility to the programmers.

Some advantages of *default parameters* are listed below :

- They can be used to add new parameters to the existing functions.
- They can be used to combine similar functions into one.

NOTE

The default values for parameters are considered only if no value is provided for that parameter in the function call statement.

3.5.3 Keyword (Named) Arguments

The default arguments give you flexibility to specify the default value for a parameter so that it can be skipped in the function call, if needed. However, still you cannot change the order of the arguments in the function call ; you have to remember the correct order of the arguments.

To have complete control and flexibility over the values sent as arguments for the corresponding parameters, Python offers another type of arguments : *keyword arguments.*

Python offers a way of writing function calls where **you can write any argument in any order provided you name the arguments** when calling the function, as shown below :

```
interest (prin = 2000, time = 2, rate = 0.10)
interest (time = 4, prin = 2600, rate = 0.09)
interest (time = 2, rate = 0.12, prin = 2000)
```

All the above function calls are valid now, even if the order of arguments does not match the order of parameters as defined in the function header.

In the 1st function call above,

prin gets value 2000, ***time*** gets value as 2 and ***rate*** as 0.10.

In the 2nd function call above,

prin gets value 2600, ***time*** gets value as 4 and ***rate*** as 0.09.

In the 3rd function call above,

prin gets value 2000, ***time*** gets value as 2 and ***rate*** as 0.12.

This way of specifying names for the values being passed, in the function call is known as **keyword arguments**.

KEYWORD ARGUMENTS

Keyword arguments are the named arguments with assigned values being passed in the function call statement.

Scanned with CamScanner

3.5.4 Using Multiple Argument Types Together

Python allows you to combine multiple argument types in a function call. Consider the following function call statement that is using both *positional (required)* and *keyword arguments* :

```
interest (5000, time = 5)
```

The first argument value (5000) in above statement is representing a positional argument as it will be assigned to first parameter on the basis of its position. The second argument (time = 5) is representing *keyword argument* or *named argument*. The above function call also skips an argument (rate) for which a default value is defined in the function header.

Rules for combining all three types of arguments

Python states that in a function call statement :

- an argument list must first contain *positional (required) arguments* followed by any *keyword argument*.
- *Keyword arguments* should be taken from the *required arguments* preferably.
- You cannot specify a value for an argument more than once.

> **NOTE**
> Having a positional arguments after keyword arguments will result into error.

For instance, consider the following function header :

```
def interest( prin, cc, time = 2, rate = 0.09) :
    return prin * time * rate
```

It is clear from above function definition that values for parameters ***prin*** and ***cc*** can be provided either as positional arguments or as keyword arguments but these values cannot be skipped from the function call.

Now for above function, consider following call statements :

Function call statement	**Legal / illegal**	**Reason**
`interest(prin = 3000, cc = 5)`	legal	non-default values provided as named arguments
`interest(rate = 0.12, prin = 5000, cc = 4)`	legal	keyword arguments can be used in any order and for the argument skipped, there is a default value
`interest(cc = 4, rate = 0.12, prin = 5000)`	legal	with keyword arguments, we can give values in any order
`interest(5000, 3, rate = 0.05)`	legal	positional arguments before keyword argument; for skipped argument there is a default value
`interest(rate = 0.05, 5000, 3 )`	illegal	keyword argument before positional arguments
`interest(5000, prin = 300, cc =2)`	illegal	Multiple values provided for ***prin*** ; once as positional argument and again as keyword argument
`interest(5000, principal = 300, cc = 2 )`	illegal	undefined name used (***principal*** is not a parameter)
`interest(500, time = 2, rate = 0.05 )`	illegal	A required argument (*cc*) is missing.

Scanned with CamScanner

Now consider the following program that creates and uses the function interest(), we have been discussing so far.

Program 3.2 Program to calculate simple interest using a function interest( ) that can receive principal amount, time and rate and returns calculated simple interest. Do specify default values for rate and time as 10% and 2 years respectively.

```
def interest(principal, time = 2, rate = 0.10) :
    return principal * rate * time

# __main__
prin = float(input("Enter principal amount :"))
print("Simple interest with default ROI and time values is :")
si1 = interest(prin)
print("Rs.", si1)
roi = float(input("Enter rate of interest (ROI) :"))
time = int(input("Enter time in years :"))
print("Simple interest with your provided ROI and time values is :")
si2 = interest(prin, time, roi/100)
print("Rs.", si2)
```

Sample run of above program is as shown below :

```
Enter principal amount : 6700
Simple interest with default ROI and time values is :
Rs. 1340.0
Enter rate of interest (ROI) : 8
Enter time in years : 3
Simple interest with your provided ROI and time values is :
Rs. 1608.0
```

3.6 RETURNING VALUES FROM FUNCTIONS

Functions in Python may or may not return a value. You already know about it. There can be broadly *two* types of functions in Python :

- Functions returning some value (*non-void functions*)
- Functions not returning any value (*void functions*)

1. Functions returning some value (Non-void functions)

The functions that return some computed result in terms of a value, fall in this category. The computed value is returned using **return** statement as per syntax :

```
return <value >
```

Scanned with CamScanner

The value being returned can be one of the following :

- a literal
- a variable
- an expression

For example, following are some legal **return** statements :

```
return 5                # literal being returned
return 6+4              # expression involving literals being returned
return a                # variable being returned
return a**3             # expression involving a variable and literal, being returned
return (a + 8**2) / b   # expression involving variables and literals, being returned
return a + b /c         # expression involving variables being returned
```

When you call a function that is returning a value, the returned value is made available to the caller function/program by internally substituting the function call statement. Confused ? Well, don't be. Just read on, please ☺ .

Suppose if we have a function :

```
def sum (x, y) :
    s = x + y
    return s
```

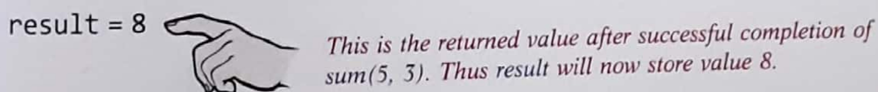
And we are invoking this function as :

```
result = sum(5, 3)
```

The returned value from sum( ) will replace this function call.

After the function call to **sum( )** function is successfully completed, (*i.e.*, the return statement of function has given the computed sum of 5 and 3) **the returned value** (8 in our case) **will internally substitute the function call** statement. That is, now the above statement will become (internally) :

```
result = 8
```



This is the returned value after successful completion of sum(5, 3). Thus result will now store value 8.

IMPORTANT

- **The returned value of a function should be used in the caller function/program inside an expression or a statement** *e.g.*, for the above mentioned *sum( )* function, following statements are using the returned value in right manner :

```
add_result = sum (a, b)     # The returned valued being used in assignment statement
print(sum(3, 4))            # The returned valued being used in print statement
sum (4, 5) > 6              # The returned valued being used in a relational expression
```

If you do not use their value in any of these ways and just give a stand-alone function call, **Python will not report an error but their return value is completely wasted.**

NOTE

Functions returning a value are also known as **fruitful functions**.

Scanned with CamScanner

- **The return statement ends a function execution even if it is in the middle of the function.** A function ends the moment it reaches a *return* statement or all statements in function-body have been executed, whichever occurs earlier, *e.g.*, following function will never reach print( ) statement as *return* is reached before that.

```
def check (a) :
    a = math.fabs(a)
    return a
    print(a)
check(-15)
```

This statement is unreachable because check() function will end with return and control will never reach this statement

Caution! If you do not use a function call of a function-returning some value inside any other expression or statement, function will be executed but its return value will be wasted ; Python will not report any error for it.

2. Functions not returning any value (Void functions)

The functions that perform some action or do some work but do not return any computed value or final value to the caller are called **void functions**. A void function may or may not have a return statement. If a *void function* has a return statement, then it takes the following form :

```
return
```

For a void function, return statement does not have any value/expression being returned.

that is, keyword **return** without any value or expression. Following are some examples of void functions :

void function but no return statement

```
def greet( ):
    print("helloz")
```

Another void function with no return statement

```
def greet1(name) :
    print("hello", name)
```

void function with a return statement

```
def quote( ) :
    print("Goodness counts!!")
    return
```

```
def prinSum (a, b, c) :
    print("Sum is", a + b + c)
    return
```

Another void function with a return statement

The void functions are generally not used inside a statement or expression in the caller ; their function call statement is standalone complete statement in itself, *e.g.*, for the all four above defined *void functions*, the function-call statements can take the form :

```
greet( )
greet1( )
quote( )
prinSum(4, 6)
```

As you can see that all these function call statements are standalone, i.e., these are not part of any other expression or statement

The void functions do not return a value but they return a legal empty value of Python *i.e.*, **None**. Every void function returns value **None** to its caller. So if need arises you can assign this return value somewhere as per your needs, *e.g.*, consider following program code :

Scanned with CamScanner

```
def greet( ):
    print("helloz")

a = greet( )
print(a)
```

> **NOTE**
>
> A void function (sometimes called **non-fruitful functions**) returns legal empty value of Python *i.e.*, **None** to its caller.

The above program will give output as :

```
helloz
None
```

Yes, you guessed it right – **helloz** is printed because of *greet( )*'s execution and **None** is printed as value stored in **a** because *greet*( ) returned value **None**, which is assigned to variable **a**.

Consider the following example :

```
# Code 1
def replicate() :
    print("$$$$$")

print(replicate())
```

```
# Code 2
def replicate1() :
    return "$$$$$"

print(replicate1())
```

Here the outputs produced by above two codes will be :

Outputs :

Code 1

```
$$$$$
None
```

Code 2

```
$$$$$
```

I know that you know the reason, why ?

So, now you know that in Python you can have following *four* possible combinations of functions :

(*i*) non-void functions without any arguments

(*ii*) non-void functions with some arguments

(*iii*) void functions without any arguments

(*iv*) void functions with some arguments

Please note that a function in a program can call any other function in the same program.

3.6.1 Returning Multiple Values

Unlike other programming languages, Python lets you return more than one value from a function. Isn't that useful ? You must be wondering, how ? Let's find out.

To return multiple values from a function, you have to ensure following things :

(*i*) The return statement inside a *function body* should be of the form given below :

```
return <value1/variable1/expression1>, <value2/variable2/expression2>, ...
```

(*ii*) The *function call statement* should receive or use the returned values in one of the following ways :

Scanned with CamScanner

larger expression; or a statement as a part of larger statement.

3.8 SCOPE OF VARIABLES

The scope rules of a language are the rules that decide, in which part(s) of the program, a particular piece of code or data item would be known and can be accessed therein. To understand **Scope**, let us consider a real-life situation.

Suppose you are touring a historical place with many monuments. To visit a monument, you have to buy a ticket. Say, you buy a ticket (let us call it *ticket1*) to go see a *monumentA*. As long as, you are inside *monumentA*, your ticket1 is valid. But the moment you come out of *monumentA*, the validity of *ticket1* is over. You cannot use *ticket1* to visit any other monument. To visit *monumentB*, you have to buy another ticket, say *ticket2*. So, we can say that scope of *ticket1* is *monumentA* and scope of *ticket2* is *monumentB*.

Say, to promote tourism, the government has also launched a city-based ticket (say *ticket3*). A person having city-based ticket can visit all the monuments in that city. So we can say that the scope of *ticket3* is the whole city and all the monuments within city including *monumentA* and *monumentB*.

Now let us understand scope in terms of Python. In programming terms, we can say that, **scope** refers to part(s) of program within which a name is legal and accessible. If it seems confusing, I suggest you read on the following lines and examples and then re-read this section.

SCOPE

Part(s) of program within which a name is legal and accessible, is called scope of the name.

There are broadly *two* kinds of scopes in Python, as being discussed below.

1. Global Scope

A name declared in top level segment (__main__) of a program is said to have a **global scope** and is *usable inside the whole program* and all blocks (functions, other blocks) contained within the program.

(Compare with real-life example given above, we can say that ticket3 has global scope within a city as it is usable in all blocks within the city.)

Scanned with CamScanner

2. Local Scope

A name declared in a function-body is said to have **local scope** *i.e.*, it can be *used only within this function and the other blocks contained under it*. The *names* of *formal arguments* also have local scope.

(*Compare with real-life example given above, we can say that ticket1 and ticket2 have local scopes within* ***monumentA*** *and* ***monumentB*** *respectively.*)

> **NOTE**
>
> A *global variable* is a variable defined in the 'main' program (__main__ section). Such variables are said to have *global scope*.
>
> A *local variable* is a variable defined within a function. Such variables are said to have *local scope*.

A local scope can be multi-level; there can be an enclosing local scope having a nested local scope of an inside block. All this would become clear to you in coming lines.

Scope Example I

Consider the following Python program (program 3.1 of section 3.4) :

```
1.  def calcSum (x, y) :
2.        z = x + y              # statement -1-
3.        return z               # statement -2-

4.  num1 = int( input( "Enter first number :" ) )        # statement -1-
5.  num2 = int( input( "Enter second number :" ) )       # statement -2-
6.  sum = calcSum ( num1, num2 )                         # statement -3-
7.  print('Sum of given numbers is', sum)                # statement -4-
```

A careful look on the program tells that there are *three* variables ***num1, num2*** and ***sum*** defined in the *main program* and three variables *x*, *y* and *z* defined in the function ***calcSum( )***. So, as per definition given above, ***num1, num2*** and ***sum*** are **global variables** here and *x*, *y* and *z* are local variables (local to function ***calcSum( )***).

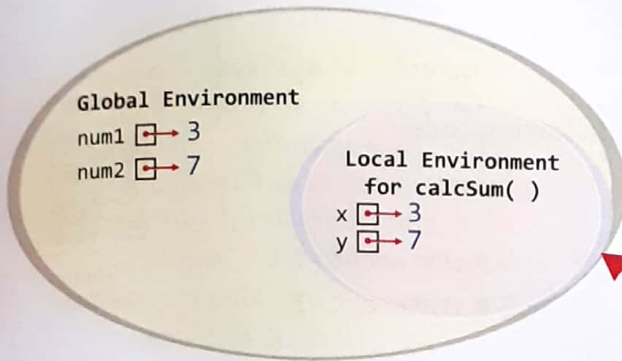
Let us now see how there would be different scopes for variables in this program by checking the status after every statement executed. We'll check the status as per the flow of execution of above program (refer to section 3.4)

1. Line1 : *def* encountered and lines 2-3 are ignored.

2. Line4 (**Main.1**) : Execution begins and global environment is created. ***num1*** is added to the environment.

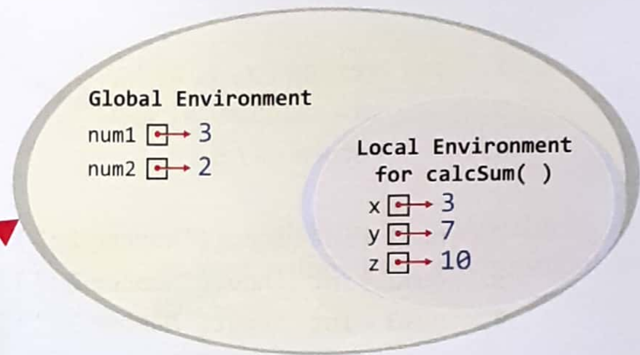
3. Line5 (**Main.2**) : ***num2*** is also added to the global environment.

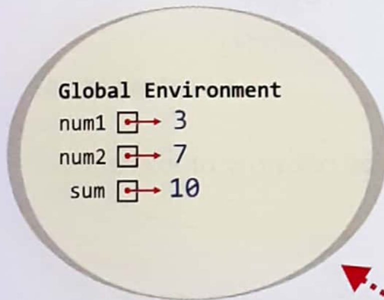
Scanned with CamScanner



4. Line6 (**Main.3**) : **calcSum( )** is invoked, so a local environment for **calcSum( )** is created; *formal arguments* *x* and *y* are created in local environment.

5. Line2 (**calcSum.1**) : variable *z* is created in the local environment.





6. Line3 (**calcSum.2**) : value of *z* is returned to caller (**return** ends the function, hence after sending value of ***s*** to caller in variable ***sum*** (when control is back to ***Main.3***), the local environment is removed and so are all its constituents).

7. Line7 (**Main.4**) : the print statement picks value of ***sum*** from its own environment.

8. Program over. Global environment is also removed with the end of the program.

As you can see from above that scope of names ***num1***, ***num2*** and ***sum*** is **global** and scope of names *x*, *y* and *z* is **local**.

Variables defined outside all functions are global variables

These variables can be defined even before all the function definitions.

Consider the following example :

```
x = 5
def func(a):
    b = a + 1
    return b

y = input( "Enter number" )
z = y + func(x)
print(z)
```

*Variable **x** defined above all functions. It is also a global variable along with y and z*

Scanned with CamScanner

To see
Variable Scope
in action

Scan
QR Code

Scope Example 2

Let us take one more example. Consider the following code :

```
1.  def calcSum(a, b, c) :
2.      s = a + b + c                           # statement -1-
3.      return s                                # statement -2-

4.  def average ( x, y, z) :
5.      sm = calcSum (x, y, z)                  # statement -1-
6.      return sm / 3                           # statement -2-

7.  num1 = int (input( "Number 1 :" ) )         # statement -1-
8.  num2 = int (input( "Number 2 :" ) )         # statement -2-
9.  num3 = int (input( "Number 3 :" ) )         # statement -3-
10. print("Average of these numbers is", average( num1, num2, num3))
                                                        # statement -4-
```

Internally the global and local environments would be created as per flow of execution :

1. Line1 : *def* encountered ; lines 2, 3 ignored.
2. Line4 : *def* encountered ; lines 5, 6 ignored.
3. Line7 (**Main.1**) : execution of main program begins ; global environment created ; ***num1*** added to it.

Global Environment
num1 → 3

4. Lines 8, 9 (**Main.2** and **Main.3**) : add ***num2*** and ***num3*** to global environment.

Global Environment
num1 → 3
num2 → 7
num3 → 5

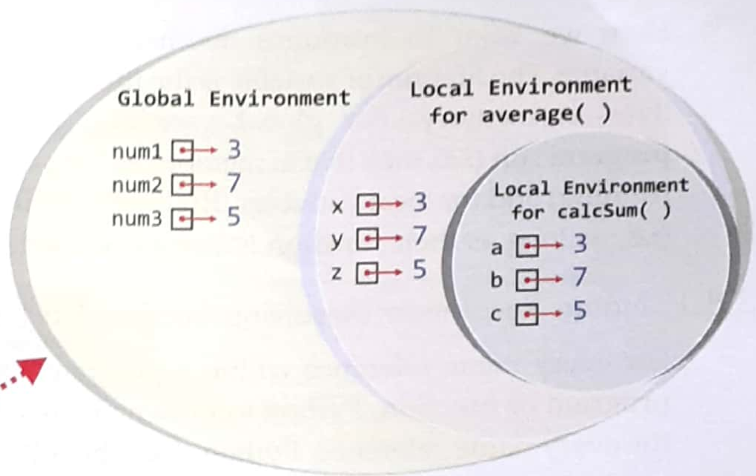
5. Line10 (**Main.4**) : Function **average( )** is invoked, so a local environment for **average( )** is created ; formal arguments *x*, *y* and *z* are created in local environment.

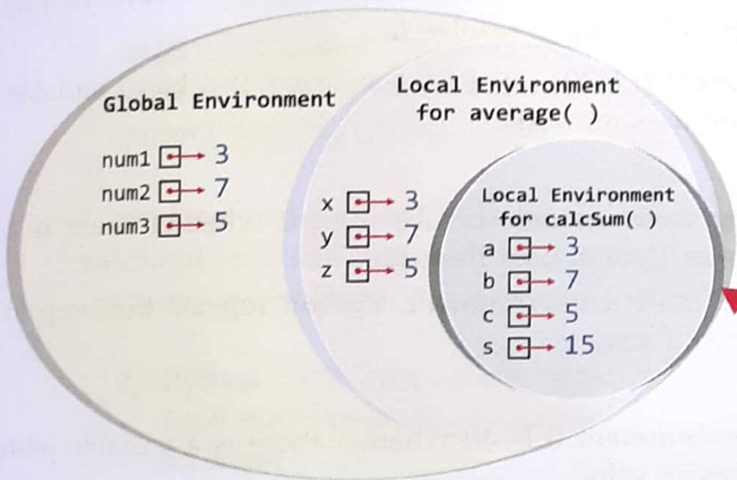
Global Environment
num1 → 3
num2 → 7
num3 → 5

Local Environment for average( )
x → 3
y → 7
z → 5

Scanned with CamScanner

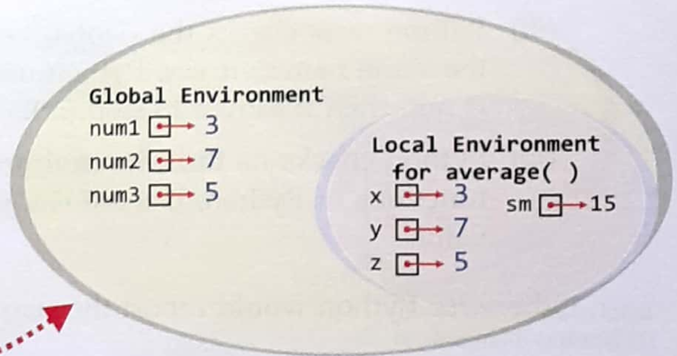
6. Line5 (**average.1**) : Function *calcSum( )* is invoked, so a local environment for *calcSum( )* is created, nested within local environment of *average( )* ; its formal arguments (*a*, *b*, *c*) are created in it.



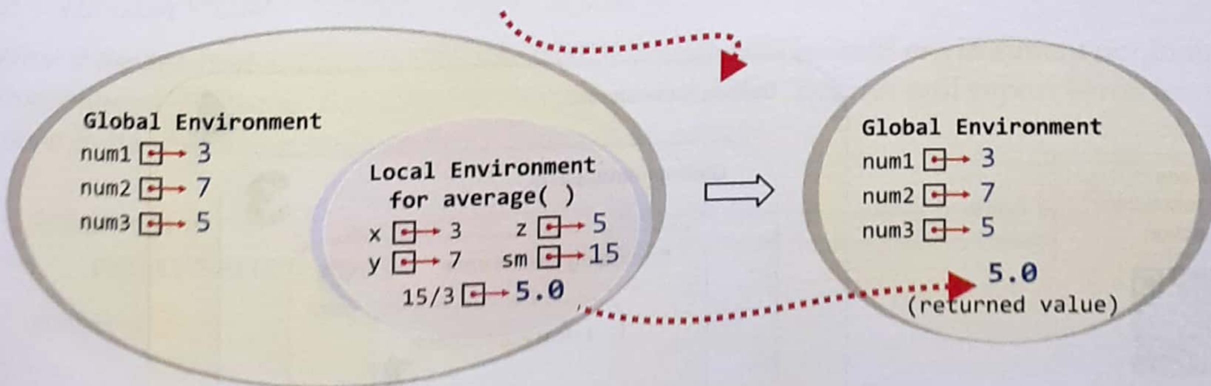


7. Line1 (**calcSum.1**) : Variable *s* is created within local environment of *calcSum( )*.

8. Line2 (**calcSum.2**) : Value of *s* is returned to *sm* of *average*( ) and *calcSum*( ) is over, hence the local environment of *calcSum( )* is removed.



9. Line6 (**average.2**) : Return value is calculated as *sm* / **3** (*i.e.*, 15/3 = 5.0) and returned to caller (*main.4*) statement ; *average( )* is over so its local environment is removed.



10. Line10 (**Main.4**) : The *print statement* receives computed value 5.0, prints it and program is over. (with this global environment of the program will also be removed.)

What if you want to use the global variable inside local scope?

If you want to use the value of already created global variable inside a local function without modifying it, then simply use it. Python will use LEGB rule and reach to this variable.

But if you want to assign some value to the global variable without creating any local variable, then what to do ? This is because, if you assign any value to a name, Python will create a local variable by the same name. For this kind of problem, Python makes available **global** statement.

> **NOTE**
>
> The **global** statement is a declaration which holds for the entire current code block. It means that the listed identifiers are part of the *global namespace* or *global environment*.

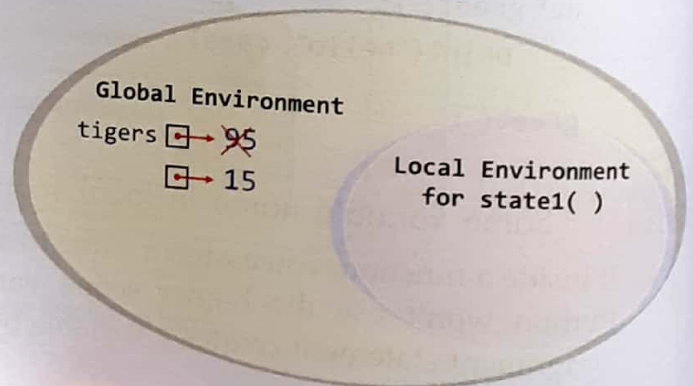
To tell a function that for a particular name, do not create a local variable but use global variable instead, you need to write :

```
global <variable name>
```

For example, in above code, if you want function state1( ) to work with global variable **tigers**, you need to add global statement for **tigers** variable to it as shown below :

```
def state1( ) :
    global tigers
    tigers = 15
    print(tigers)
tigers = 95
print(tigers)
state1()
print(tigers)
```

*This is an indication not to create local variable with the name **tigers**, rather use global variable **tigers**.*



Scanned with CamScanner

The above program will give output as :

```
95
15
15
```

*Result of print statement inside state1( ) function, value of global **tigers** is printed (which was modified to **15** in previous line).*

*Result of print statement inside main program, thus, value of global **tigers** (which is 15 now) is printed.*

Once a variable is declared *global in a function, you cannot undo the statement*. That is, after a global statement, the function will always refer to the global variable and local variable cannot be created of the same name.

But for good programming practice, the use of global statement is always discouraged as with this programmers tend to lose the control over variables and their scopes.

NOTE

The **global** statement cannot be reverted in a program run. One should avoid using **global** statement in Python program.

TIP

Although global variables can be accessed through local scope, but it is not a good programming practice. So, keep global variables global, and local variables local.

Scanned with CamScanner

3.9.1 Mutability/Immutability of Arguments/Parameters and Function Calls

When you pass values through arguments and parameters to a function, mutability/ immutability also plays an important role there.

Let us understand this with the help of some sample codes.

Sample Code 1.1

Passing an Immutable Type Value to a function.

```
1.  def myFunc1(a):
2.      print("\t Inside myFunc1()")
3.      print("\t Value received in 'a' as", a)
4.      a = a + 2
5.      print("\t Value of 'a' now changes to", a)
6.      print("\t returning from myFunc1()")
7.  #__main__
8.  num = 3
9.  print("Calling myFunc1() by passing 'num' with value", num)
10. myFunc1(num)
11. print("Back from myFunc1(). Value of 'num' is", num)
```

Now have a look at the output produced by above code as shown below :

```
Calling myFunc1( ) by passing 'num' with value 3
    Inside myFunc1( )
    Value received in 'a' as 3
    Value of 'a' now changes to 8
    returning from myFunc1( )
Back from myFunc1( ). Value of 'num' is 3
```

The value got changed from 3 to 8 inside function BUT NOT got reflected to __main__

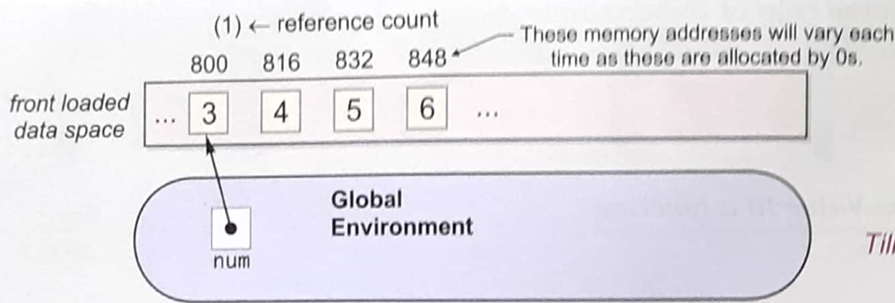
As you can see that the function ***myFunc1()*** received the passed value in parameter ***a*** and then changed the value of ***a*** by performing some operation on it. Inside **myFunc1( )**, the value (of ***a***) got changed but after returning from **myFunc1( )**, the originally passed variable ***num*** remains unchanged.

Let us see how the memory environments are created for above code (*i.e.*, ***sample code1***).

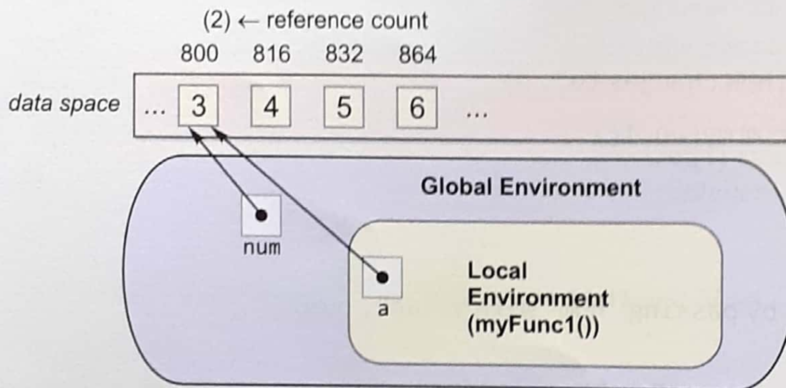
Scanned with CamScanner

Memory Environment For Sample Code 1.1

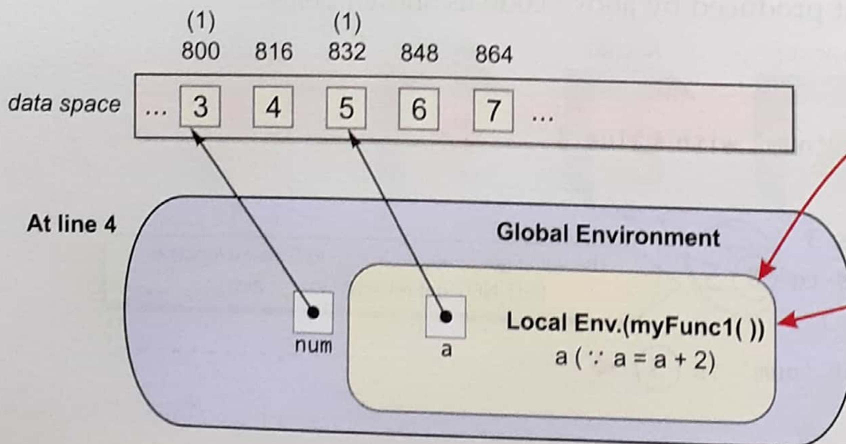
(Note : Passed value is an integer, an immutable type)



Till Lines 7-9 of code of --main--

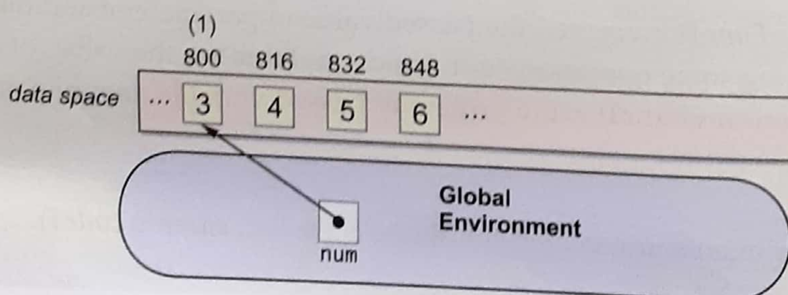


*At **line10** (function is called) argument num is received in parameter **a** and for **lines 1, 2, 3**, environment remains the same*



*See, the global environment's **num** remains unaffected from changes to variable **a** of myfunc1*

*Local memory environment remains the same till **line6** and the **myfunc1( )** gets over and control returns to --main-- part's **line11** and the local environment of myfunc1( ) is removed.*



*At **line11**, when **num's** value is printed Python prints 3 as the num remained unchanged in its scope and thus always remained 3.*

Scanned with CamScanner

So, you just saw how Python processed an immutable data type when it is passed as argument. Let us see what happens inside memory if you pass a mutable type such as a ***list***. (Recall that a sequence/collection such as a list internally is stored as a container that holds the references of individual items.)

Sample Code 2.1

Passing a Mutable Type Value to a function–Making changes in place)

```
1.  def myFunc2(myList):
2.       print("\n\t Inside CALLED Function now")
3.       print("\t List received:", myList)
4.       myList[0] += 2
5.       print("\t List within called function, after changes:", myList)
6.       return

7.  List1 = [1]
8.  print("List before function call : ", List1)
9.  myFunc2(List1)
10. print("\nList after function call : ", List1)
```

Now have a look at the output produced by above code as shown below :

```
List before function call : [1]
  Inside CALLED Function now
  List received:  [1]
  List within called function, after changes : [3]

List after function call : [3]
```

The value got changed from [1] to [3] inside function and change GOT REFLECTED to __main__

As you can see that the function myFunc2( ) receives a mutable type, ***a list***, this time. The passed list (**List1**) contains value as [1] and is received by the function in parameter **mylist**. The changes made inside the function in the list ***mylist*** get reelected in the original list passed, *i.e.*, in list1 of __main__ .

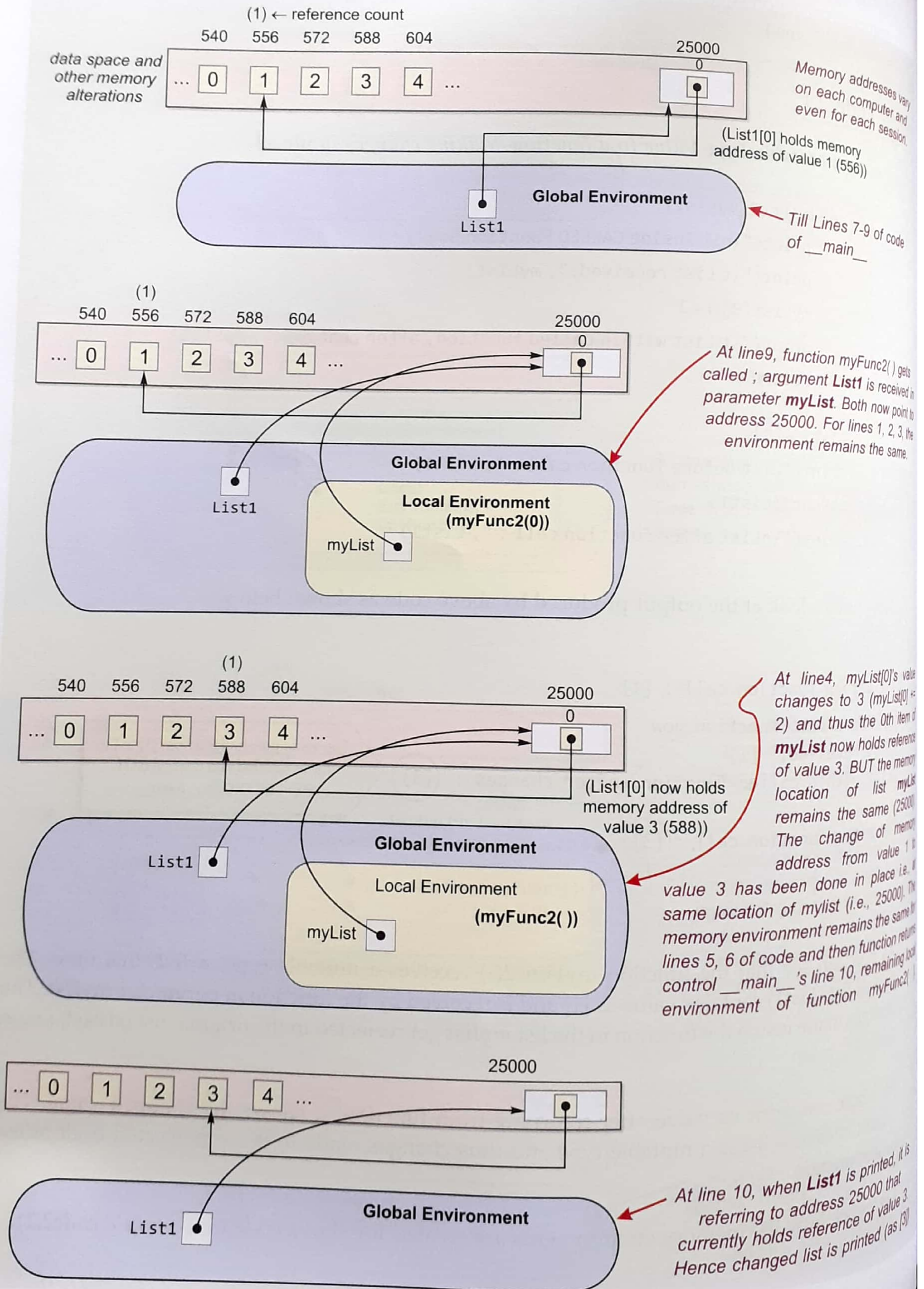
So when you print its value after returning from function, it shows the changed value. The reason is clear – list is a mutable type and thus changes made to it are refelected back in the caller function.

Let us see how the memory environments are created for above code (*i.e.*, ***sample code2.1***).

Scanned with CamScanner

Memory Environment For Sample Code 2.1

(Note : Passed value is a *list*, a mutable type)



Scanned with CamScanner

LET US REVISE

- *A* **Function** *is a subprogram that acts on data and often returns a value.*
- *Functions make program handling easier as only a small part of the program is dealt with at a time, thereby avoiding ambiguity.*
- *By default, Python names the segment with top-level statements (main program) as* **__main__** *.*
- *A Function is executed in an* **execution frame**.
- *The values being passed through a function-call statement are called* **arguments** *(or* **actual parameters** *or* **actual arguments**).
- *The values received in the function definition/header are called* **parameters** *(or* **formal parameters** *or* **formal arguments**).
- *Python supports three types of formal arguments : parameters (i) Positional arguments (Required arguments), (ii) Default arguments and (iii) Keyword (or named) arguments.*
- *When the function call statement must match the number and order of arguments as defined in the function definition, this is called the* **positional argument matching**.
- *A parameter having default value in the function header is known as a* **default parameter**.
- *A default argument can be skipped in the function call statement.*
- *The default values for parameters are considered only if no value is provided for that parameter in the function call statement.*
- **Keyword arguments** *are the named arguments with assigned values being passed in the function call statement.*
- *A function may or may not return a value.*
- *A function may also return multiple values that can either be received in a tuple variable or equal number of individual variables.*
- *A function that returns a non-empty value is a non-void function.*
- *Functions returning value are also known as* **fruitful functions**.
- *A function that does not return a value is known as void function or* **non-fruitful function**.
- *A void function internally returns legal empty value* **None**.
- *A function in a program can invoke any other function of that program.*
- *The program part(s) in which a particular piece of code or a data value (e.g., variable) can be accessed is known as Variable Scope.*
- *In Python, broadly scopes can either be global scope or local scope.*
- *Python resolves the scope of a name using LEGB rule, i.e., it checks environments in the order : Local, Enclosing, Global and Built-in.*
- *A local variable having the same name as that of a global variable, hides the global variable in its function.*
- *The global statement tells a function that the mentioned variable is to be used from global environment.*
- *The global statement cannot be undone in a code-block i.e., once an identifier is declared global, it cannot be reverted to local namespace.*
- *A function can also return multiple values.*
- *Mutability of arguments/parameter affects the change of value in caller function.*

Scanned with CamScanner

16. *Predict the output of the following code fragment ?*

```
def func(message, num = 1):
    print(message * num)

func('Python')
func('Easy', 3)
```

Solution.

```
Python
EasyEasyEasy
```

17. *Find and write the output of the following python code :* [CBSE Sample Paper 2019-20]

```
def fun(s):
    k = len(s)
    m = " "
    for i in range(0,k):
        if(s[i].isupper( )):
            m = m+s[i].lower( )
        elif s[i].isalpha( ):
            m = m+s[i].upper( )
        else:
            m = m+'bb'
    print(m)
fun('school2@com')
```

Solution.

```
SCHOOLbbbbCOM
```

18. *Find and write the output of the following python code :* [CBSE Sample Paper 2019-20]

```
def Change(P, Q = 30):
    P = P + Q
    Q = P - Q
    print(P,"#",Q)
    return (P)
R = 150
S = 100
R = Change(R,S)
print(R,"#",S)
S = Change(S)
```

Solution.

```
150 # 50
150 # 100
100 # 70
```

19. *Predict the output of the following code fragment ?*

```
def check(n1 = 1, n2 = 2):
    n1 = n1 + n2
```

Scanned with CamScanner

```
        n2 += 1
        print(n1, n2)

    check()
    check(2, 1)
    check(3)
```

Solution.

```
3 3
3 2
5 3
```

20. *What is the output of the following code?*

```
a = 1
def f ( ) :
    a = 10
print(a)
```

Solution. The code will print 1 to the console.

21. *What will be the output of following code?*

```
def interest (prnc, time =2 , rate = 0.10) :
    return (prnc * time * rate)

print(interest (6100, 1))
print(interest (5000, rate = 0.05))
print(interest (5000, 3, 0.12 ))
print(interest (time = 4, prnc = 5000))
```

Solution.

```
610.0
500.0
1800.0
2000.0
```

22. *Is return statement optional ? Compare and comment on the following two return statements :*

```
return
return val
```

Solution. The return statement is optional ONLY WHEN the function is *void* or we can say that when the function does not return a value. A function that returns a value, must have at least one *return* statement.

From given two return statements, statement

```
return
```

is not returning any value, rather it returns the control to caller along with empty value *None*. And the statement

```
return val
```

is returning the control to caller along with the value contained in variable *val*.

Scanned with CamScanner

23. *Write a function that takes a positive integer and returns the one's position digit of the integer.*

Solution.

```
def getOnes(num):
    # return the ones digit of the integer num
    onesDigit = num % 10
    return onesDigit
```

24. *Write a function that receives an octal number and prints the equivalent number in other number bases i.e., in decimal, binary and hexadecimal equivalents.*

Solution.

```
def oct2others( n) :
    print("Passed octal number :", n)
    numString = str(n)
    decNum = int( numString, 8)
    print("Number in Decimal :", decNum)
    print("Number in Binary :", bin(decNum))
    print("Number in Hexadecimal :", hex(decNum))

num = int(input("Enter an octal number :"))
oct2others(num)
```

Please recall that bin( ) and hex( ) do not return numbers but return the string-representations of equivalent numbers in binary and hexadecimal number systems respectively.

25. *Write a program that generates 4 terms of an AP by providing initial and step values to a function that returns first four terms of the series.*

Solution.

```
def retSeries(init, step ):
    return init, init+step, init+2*step, init+3*step

ini = int(input("Enter initial value of the AP series :"))
st = int(input("Enter step value of the AP series :"))
print("Series with initial value", ini, "& step value", st, "goes as:")
t1, t2, t3, t4 = retSeries(ini, st)
print(t1, t2, t3, t4)
```

GLOSSARY

Argument	A value provided to a function in the function call statement.
Flow of execution	The order of execution of statements during a program run.
Parameter	A name used inside a function to refer to the value which was passed to it as an argument.
Function	Named subprogram that acts on data and often returns a value.
Actual Argument	Argument
Actual Parameter	Argument
Formal Parameter	Parameter
Formal Argument	Parameter
Scope	Program part(s) in which a particular piece of code or a data value (e.g., variable) can be accessed.

Scanned with CamScanner