# COMPUTER SCIENCE With python



Textbook for
Class XII

- Programming & Computational Thinking
- Computer Networks
- Data Management (SQL, Django)
- Society, Law and Ethics

SUMITA ARORA

DHANPAT RAI & Co.

# 3 Working with Functions

85 – 140

# 3 Working with Functions

## In This Chapter





## 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.

## 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

$$f(1) = 2 \quad \text{...(1)}$$

$$f(2) = 8 \quad \text{...(2)}$$

$$f(3) = 18 \quad \text{...(3)}$$

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

- 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 :

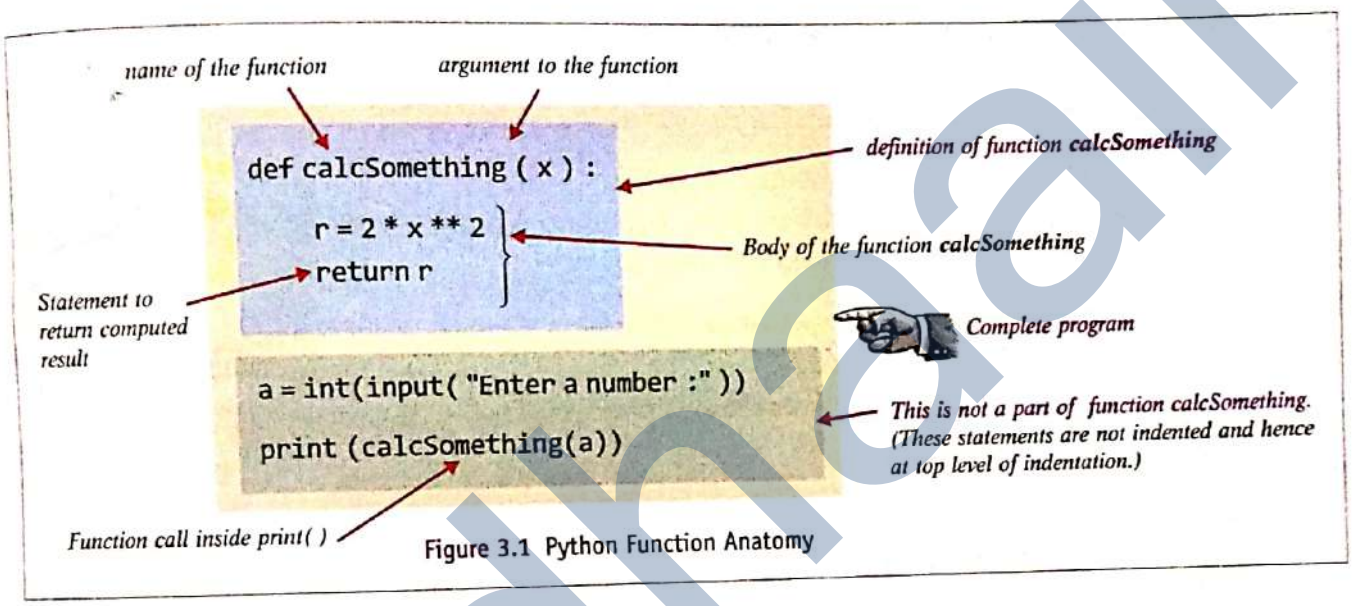



Figure 3.1 Python Function Anatomy

### 3.2.1 Calling/Inworking/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.)

Consider one more function definition given below :

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

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

```
double OfCube = 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.

## PriP STRUCTURE OF FUNCTIONS — 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.

>>>❖<<<

## 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>]
    ⋮
```

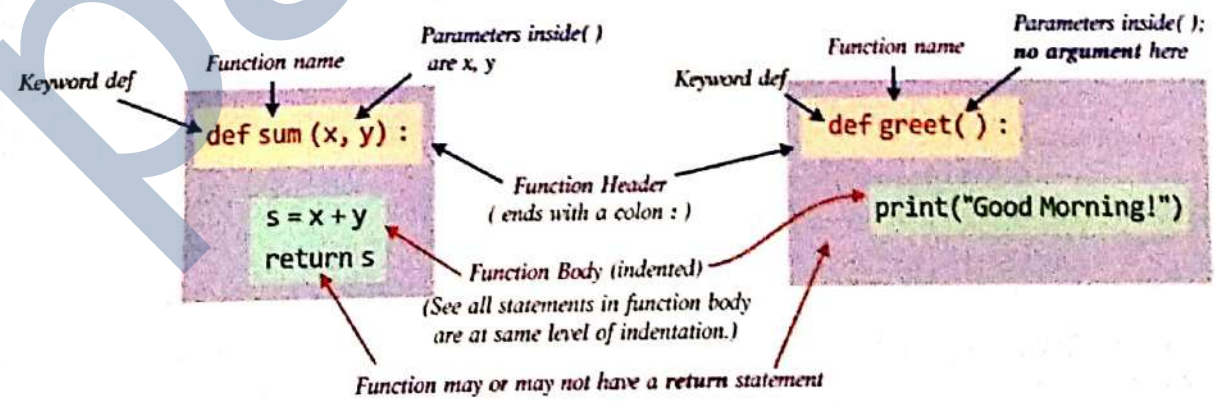
*For example*, consider some function definitions given below:

```
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*

| | |
|---|---|
| **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.<br>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.<br>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

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

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.

### 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__)*

## 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.

**FLOW OF EXECUTION**

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

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.**

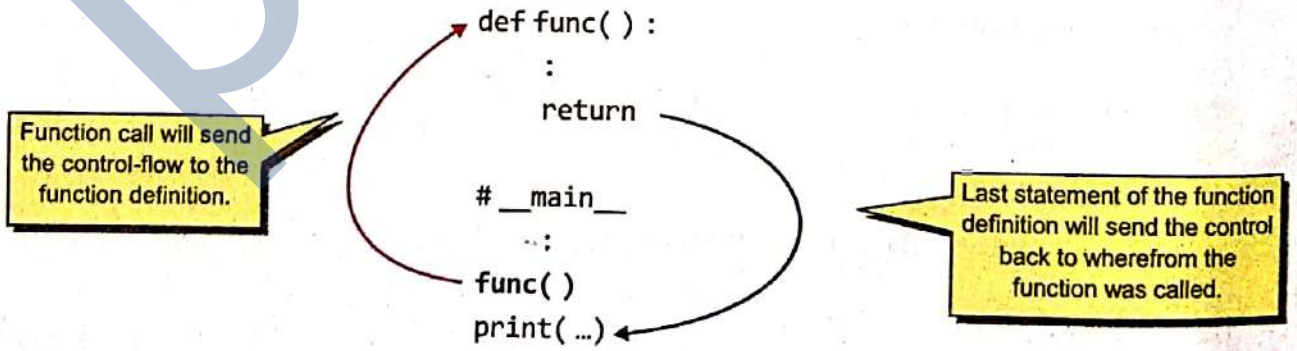
**NOTE**

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

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.

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.

## Program 3.1 *Program to add two numbers through a function*

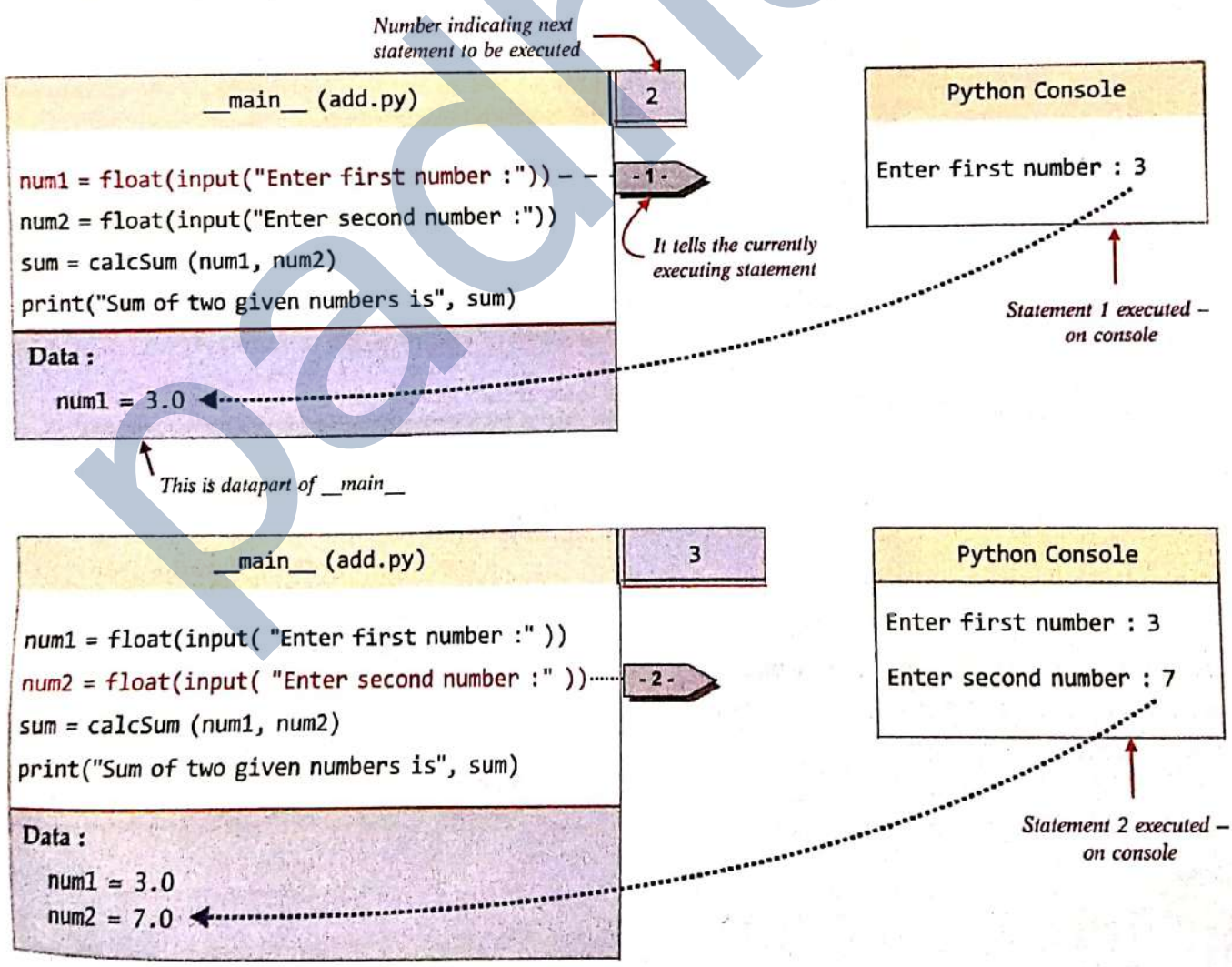
```
# program add.py to add two numbers through a function
def calcSum (x, y) :
    s = x + y                    # statement 1
    return s                     # statement 2

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)
```

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**

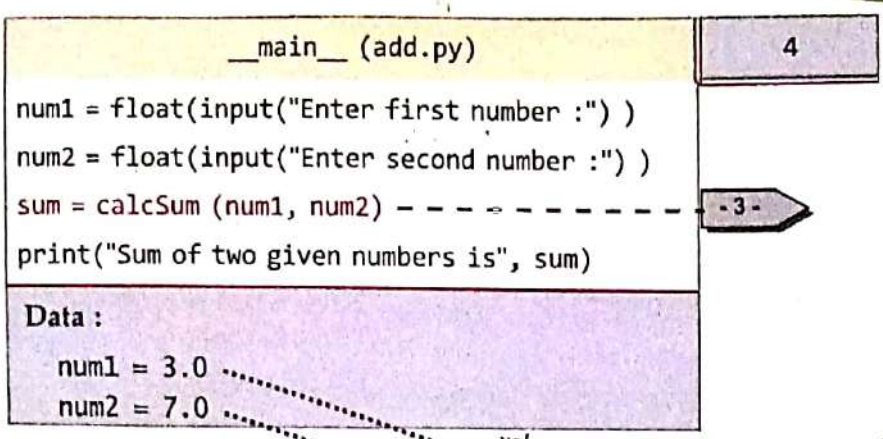
Program execution begins with first statement of *__main__* segment.

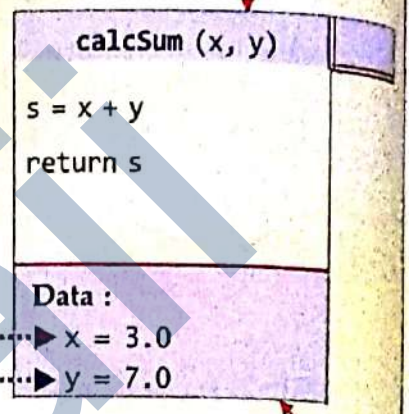


*Statement 3 is a function call statement, hence **calcSum( )**'s execution frame is created.*

*The values from __main__ are passed to it. Function **calcSum( )** receives the values in variables x and y.*

*Now the statements of **calcSum( )**'s body will be executed.*

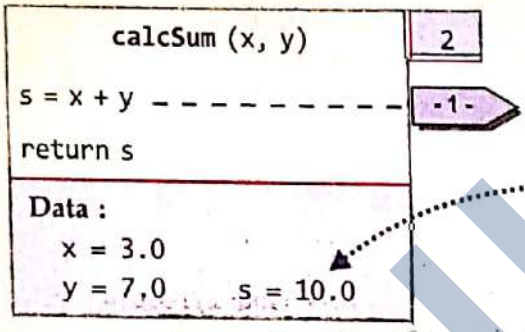


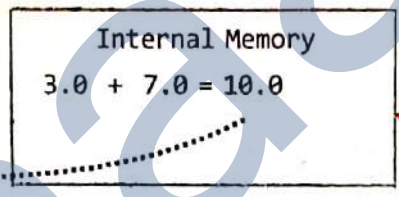




*values passed to function*

*This is data part of function calcSum( )*

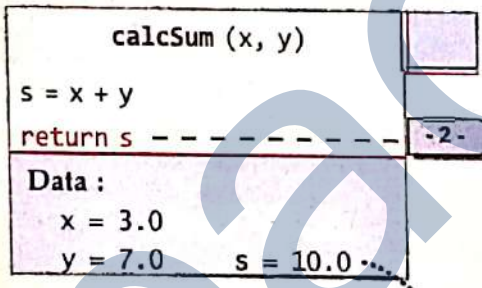
## *Function calcSum( )'s execution*



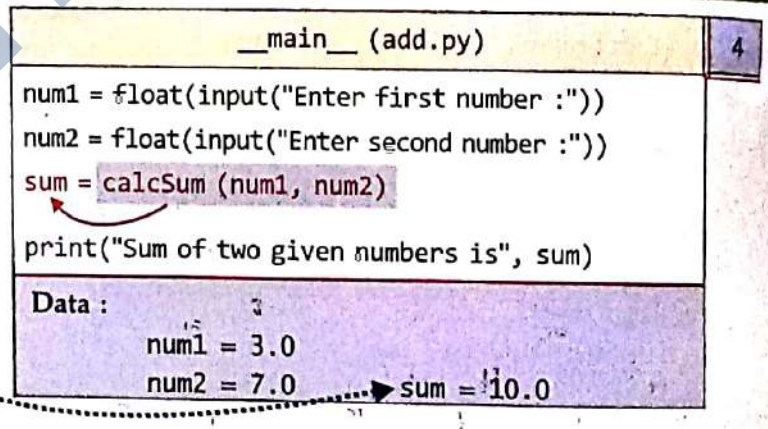



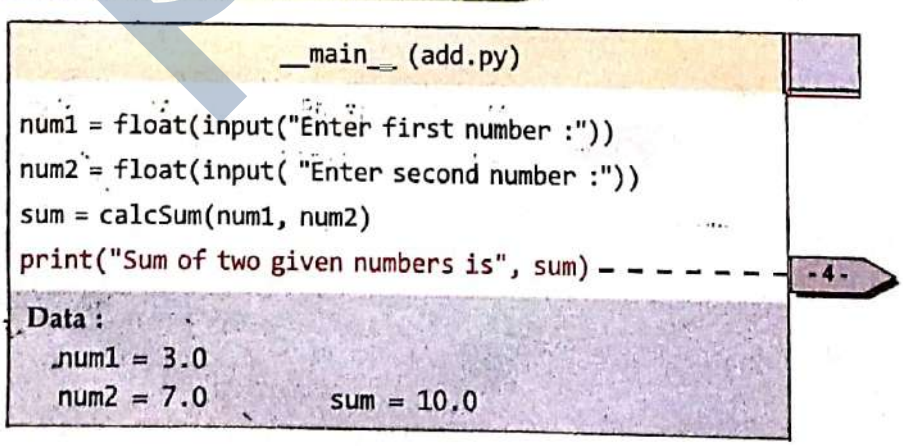

***Statement 1** of function body executed – **in memory***

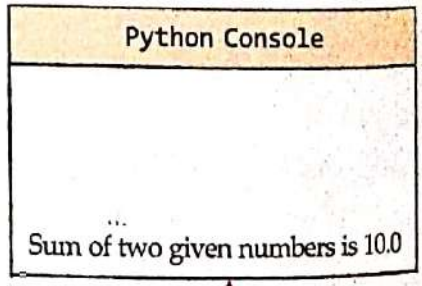
*With last statement of function body, control returns to the point wherefrom the function was called. Since the last statement of function body is a **return** statement returning value of s, value of s is given back to __main__, which stores it to variable sum. This completes the execution of **statement 3** of __main__.*




*Retrun value of calcSum( ) gets stored in **sum** variable of __main__*




*Statement 4 executed – on console*

So we can say that for above program the statements were executed as :

main.1 → main.2 → main.3 → calcSum.1 → calcSum.2 → main.3 → main.4

(As you can see that we have shown a statement as its *<segment-name>.<statement-number>*)

Now that you know how functions are executed internally, let us discuss about *actual flow of execution*.

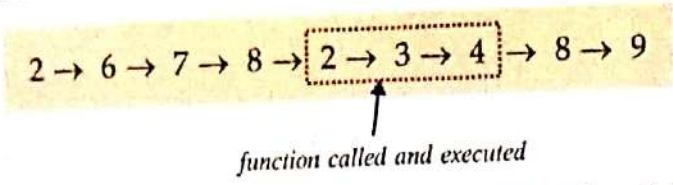
In a program, Python starts reading from line 1 downwards. Statements are executed one at a time, in order from top to bottom. While executing a program, Python follows these guidelines :

- Execution always begins at the first statement of the program.
- Comment lines (lines beginning with a #) are ignored, *i.e.*, not executed. All other non-blank lines are executed.
- If Python notices that it is a function definition, (**def statements**) then Python just executes the function header line to determine that it is proper function header and skips/ignores all lines in the function body.
- The statements inside a function-body are not executed until the function is called.
- In Python, a function can define another function inside it. But since the inner function definition is inside a function-body, the inner definition isn't executed until the outer function is called.
- When a code-line contains a *function-call*, Python first jumps to the function header line and then to the first line of the function body and starts executing it.
- A function ends with a **return** statement or the last statement of function body, whichever occurs earlier.
- If the called function returns a value *i.e.*, has a statement like return <variable/value/expression> (*e.g.*, **return a** or **return 22/7** or **return a + b** etc.) then the control will jump back to *the function call statement* and completes it (*e.g.*, if the returned value is to be assigned to variable or to be printed or to be compared or used in any type of expression etc. ; whole function call is replaced with the return value to complete the statement).
- If the called function does not return any value *i.e.*, the **return** statement has no variable or value or expression, then the control jumps back to the line following the function call statement.

If we give line number to each line in the program then flow of execution can be represented just through the line numbers, *e.g.*,

```
1.  # program add.py to add two numbers through a function
2.  def calcSum (x, y) :
3.          s = x + y           # statement 1
4.          return s            # statement 2
5.
6.  num1 = float(input("Enter first number :" ))      # 1 (statement 1)
7.  num2 = float(input("Enter second number :" ))     # 2 (statement 2)
8.  sum = calcSum (num1, num2)                        # 3 (statement 3)
9.  print("Sum of two given numbers is", sum)         # 4 (statement 4)
```

Determining flow of execution on paper is also sometimes known as *tracing the program*. As per above discussion the flow of execution for above program can also be represented as follows :




*Line 1* is ignored because it is a comment ; *line 2* is executed and determined that it is a function header, so entire function-body (*i.e., lines 3* and *4*) is ignored; *lines 6, 7* and *8* executed; *line 8* has a function call, so control jumps to the function header (*line 2*) and then to first line of function-body, *i.e., line 3*, function returns after *line 4* to line containing function call statement *i.e., lines* and then to *line 9*.

Please note that the function calling another function is called the **caller** and the function being called is the **called function** (or **callee**). In above code, the *__main__* is the caller of *calcSum( )* function.

### 3.4.1 Arguments and Parameters

As you know that you can pass values to functions. For this, you define variables to receive values in *function definition* and you send values via a *function call statement*. For *example*, consider the following program :

```
def multiply ( a, b ) :
    print (a * b)

y = 3
multiply ( 12, y )          # function call -1-
multiply ( y, y )           # function call -2-
x = 5
multiply (y, x )            # function call -3-
```

You can see that above program has a function namely **multiply( )** that receives two values. This function is being called thrice by passing different values. The *three* function calls for multiply( ) are :

```
multiply ( 12, y )          # function call -1-
multiply ( y, y )           # function call -2-
multiply (y, x )            # function call -3-
```

With *function-call 1*, the variables ***a*** and ***b*** in function header will receive values ***12*** and ***y*** respectively.
With *function-call 2*, the variables ***a*** and ***b*** in function header will receive values ***y*** and ***y*** respectively.
With *function-call 3*, the variables ***a*** and ***b*** in function header will receive values ***y*** and ***x*** respectively.

As you can see that there are **values being passed** (through function call) and **values being received** (in function definition). Let us define these *two* types of values more formally.

- *arguments* : Python refers to the values being passed as **arguments** and
- *parameters* : values being received as **parameters**.

So you can say that *arguments* appear in *function call statement* and *parameters* appear in *function header*.

Arguments in Python can be one of these value types :

- literals
- variables
- expressions

But the *parameters* in Python have to be some names *i.e.,* variables to hold incoming values.

The alternative names for **argument** are ***actual parameter*** and ***actual argument***. Alternative names for **parameter** are ***formal parameter*** and ***formal argument***. So either you use the word combination of *argument and parameter* or you can use the combination *actual parameter and formal parameter* ; or the combination *actual arguments* and *formal arguments*.

Thus for a function as defined below :

```
def multiply ( a, b ) :
    print (a * b)
```

**NOTE**

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***).

The following are some valid function call statements :

```
multiply( 3, 4 )        # both literal arguments
p = 9
multiply( p, 5)         # one literal and
                        # one variable argument
multiply( p, p + 1)     # one variable and
                        # one expression argument
```

But a function header like the one shown below is invalid :

```
def multiply ( a + 1, b ) :
    :
```

*Error!! A function header cannot have expressions. It can have just names or identifiers to hold the incoming values.*

Please remember one thing – if you are passing values of *immutable types* (*e.g., numbers, strings* etc.) to the called function then the called function cannot alter the values of passed arguments but if you are passing the values of *mutable types* (*e.g., list* or *dictionaries*) then called function would be able to make changes in them[1].

## FUNCTIONS' BASICS — Progress In Python 3.2

This *'Progress in Python'* session is aimed at strengthening the functions' basics like : *functions' terminology, function anatomy, argument vs parameter* and *flow of execution during function calls.*

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.2 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

1. This is somewhat similar to function call mechanisms, which are of *two* types : *Call by Value* and *Call by Reference*. In *Call by Value* mechanism, the called function makes a separate copy of passed values and then works with them, so original values remain unchanged. But with *Call by Reference* mechanism, the called function works with original values passed to it, thus any changes made, take place in original values only.
In Python, immutable types implement *Call By Value* mechanism and mutable types implement *Call By Reference* mechanism.

## 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

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**.

### 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*.

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.

### 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. |

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

**Program 3.2** Program to calculate simple interest using a function interest( ) that can receive principal amount, tim and rate and returns calculated simple interest. Do specify default values for rate and time as 10% an 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 >
```

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.**

- **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 :

```
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.

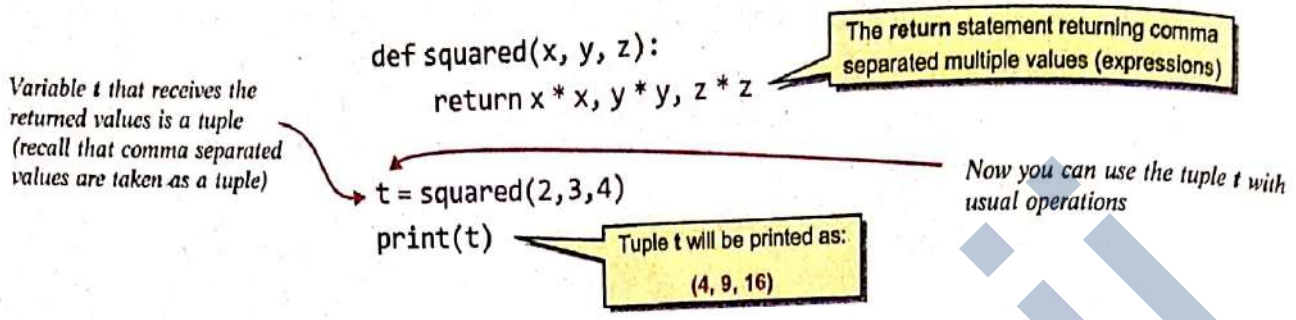
To return multiple values from a function, you have to ensure following things :

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

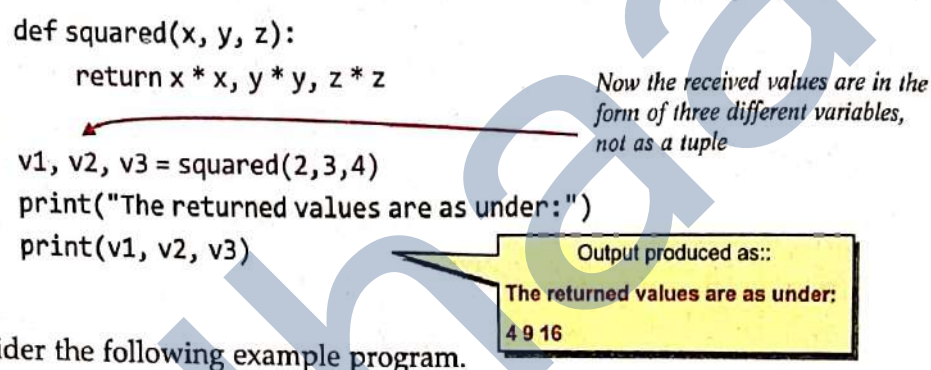
```
return <value1/variable1/expression1>, <value2/variable2/expression2>, ...
```

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

(*a*) Either receive the returned values in form a tuple variable, *i.e.*, as shown below :




(*b*) Or you can directly unpack the received values of tuple by specifying the same number of variables on the left-hand side of the assignment in function call, *e.g.*,




Now consider the following example program.

**Program 3.3** Program that receives two numbers in a function and returns the results of all arithmetic operations (+, -, *, /, %) on these numbers.

```
def arCalc(x, y) :
    return x+y, x-y, x*y, x/y, x%y

# __main__
num1 = int(input("Enter number 1 : "))
num2 = int(input("Enter number 2 : "))
add, sub, mult, div, mod = arCalc(num1, num2)
print("Sum of given numbers :", add)
print("Subtraction of given numbers :", sub)
print("Product of given numbers :", mult)
print("Division of given numbers :", div)
print("Modulo of given numbers :", mod)
```

Sample run of above program is as shown below :

```
Enter number 1 : 13
Enter number 2 : 7
Sum of given numbers : 20
Subtraction of given numbers : 6
Product of given numbers : 91
Division of given numbers : 1.8571428571428572
Modulo of given numbers : 6
```

## 3.7 COMPOSITION

Composition in general refers to using an expression as part of a larger expression; or a statement as a part of larger statement. In functions' context, we can understand composition as follows :

The arguments of a function call can be any kind of expression :

- an arithmetic expression *e.g.*,

```
greater( (4+5), (3+4) )
```

- a logical expression *e.g.*,

```
test( a or b)
```

- a function call (function composition) *e.g.*,

```
int(str(52))
int(float("52.5") * 2)
int(str(52) + str(10))
```

*Function call as part of larger function call i.e., composition*

**NOTE**

**Composition** in general refers to using an expression as part of a larger expression; or a statement as a part of larger statement.

The above examples show you composition of function calls – a function call as part of larger function call.

## 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.)*

## 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 1

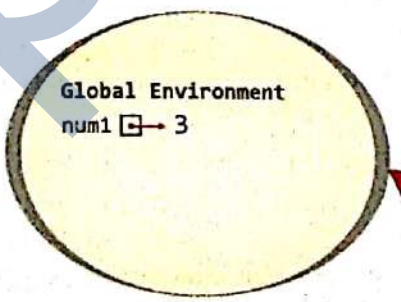
Consider the following Python program (program 3.1 of section 3.4) :

```
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-
```

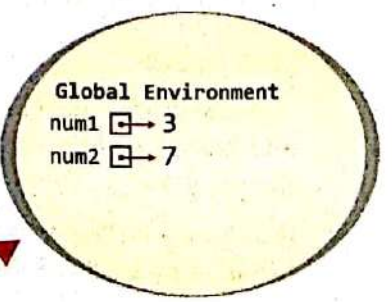
A careful look on the program tells that there are *three* variables ***num1, num2*** and ***sum*** defined in the *main program* and three variables ***x, y*** and ***z*** defined in the function ***calcSum( )***. So, as per definition given above, ***num1, num2*** and ***sum*** are **global variables**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

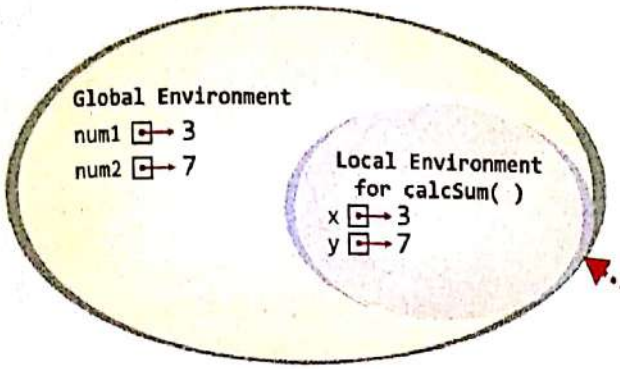
1. Line1 : *def* encountered and lines 2-3 are ignored.




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

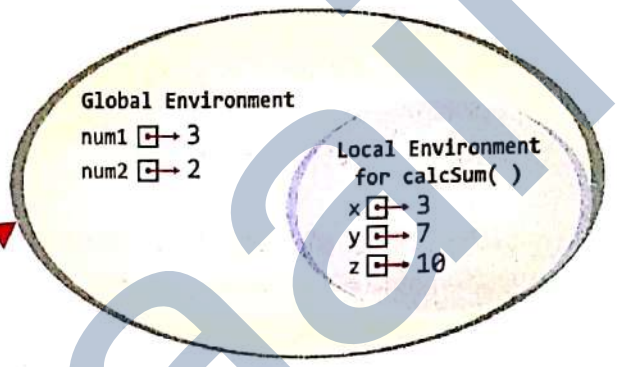


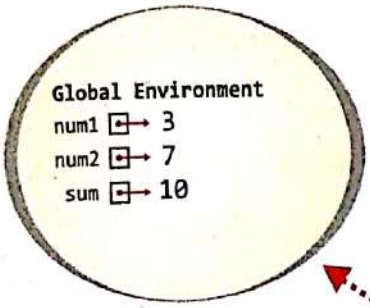

3. Line5 **(Main.2)** : ***num2*** is also added to the global environment.




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***

## Scope Example 2

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

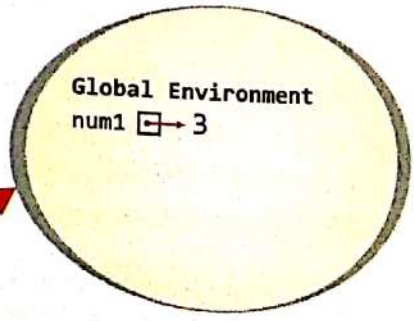
```
1.  def calcSum(a, b, c) :
2.      s = a + b + c                          # statement -1-
3.      return s                               # statement -2-

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.

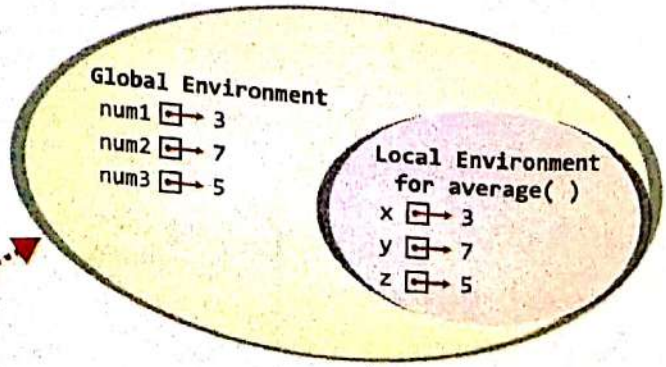
3. Line7 (**Main.1**) : execution of main program begins ; global environment created ; ***num1*** added to it.



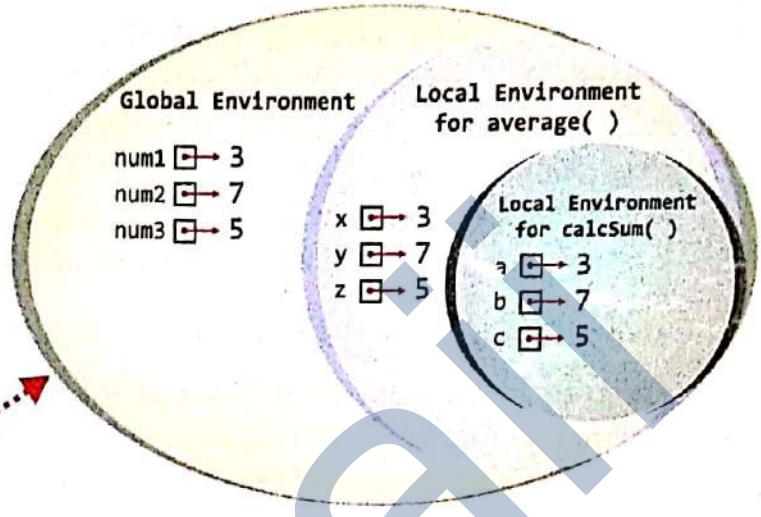


4. Lines 8, 9 (**Main.2** and **Main.3**) : add ***num2*** and ***num3*** to global environment.

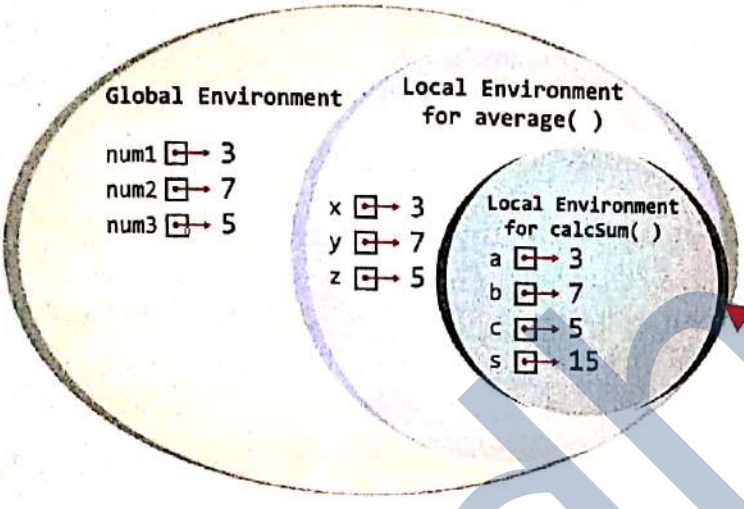
5. Line10 (**Main.4**) : Function **average( )** is invoked, so a local environment for **average( )** is created ; formal arguments *x*, *y* and *z* are created in local environment.



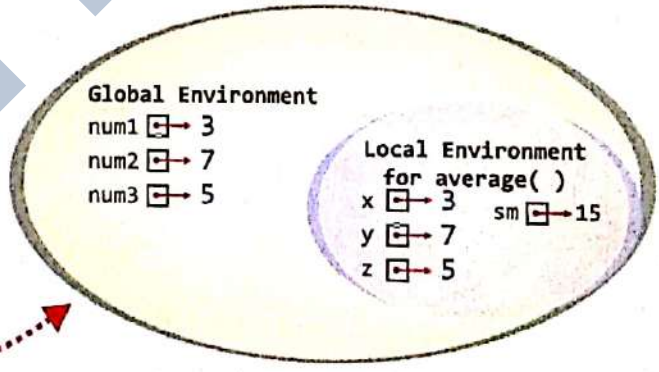
6. Line5 (**average.1**) : Function *calcSum( )* is invoked, so a local environment for *calcSum( )* is created, nested within local environment of *average( )* ; its formal arguments (*a*, *b*, *c*) are created in it.



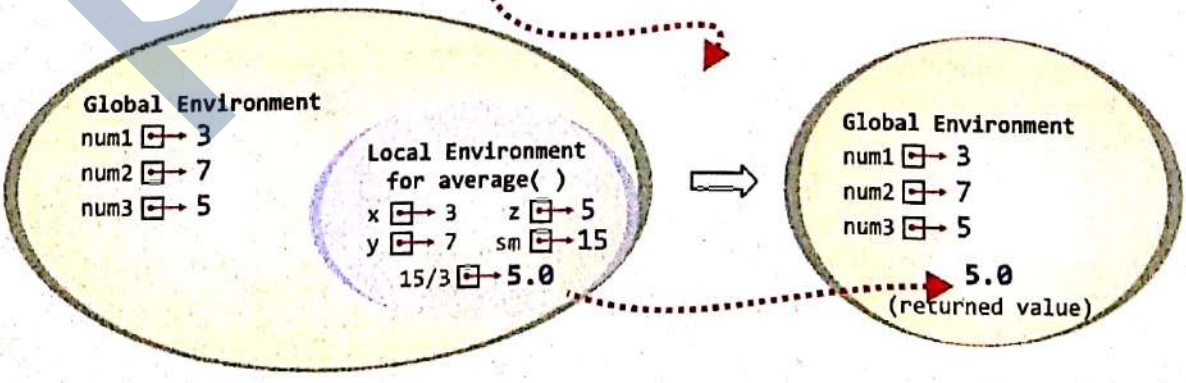

7. Line1 (**calcSum.1**) : Variable *s* is created within local environment of *calcSum( )*.




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.)

Here we want to introduce another term – **lifetime** of a variable. The lifetime of variable is the time for which a variable lives in memory. For global variables, **lifetime** is **entire program run** (*i.e.*, they live in memory as long as the program is running) and for local variables, **lifetime** is their **function's run** (*i.e.*, as long as their function is being executed.)

**LIFETIME**

The time for which a variable or name remains in memory is called Lifetime of variable.

### 3.8.1 Name Resolution (Resolving Scope of a Name)

For every name reference within a program, *i.e.*, when you access a variable from within a program or function, Python follows name resolution rule, also known as **LEGB rule**. That is, for every name reference, Python does the following to resolve it :

(*i*) It checks within its **Local environment** (LEGB) ( or *local namespace*) if it has a variable with the same name ; if yes, Python uses its value.
If not, then it moves to step (*ii*).

(*ii*) Python now checks the **Enclosing environment** (LEGB) (*e.g.*, if whether there is a variable with the same name) ; if yes, Python uses its value.
If the variable is not found in the current environment, Python repeats this step to higher level enclosing environments, if any.
If not, then it moves to step (*iii*).

(*iii*) Python now checks the **Global environment** (LEGB) whether there is a variable with the same name ; if yes, Python uses its value.
If not, then it moves to step (*iv*).

(*iv*) Python checks its **Built-in environment** (LEGB) that contains all built-in variables and functions of Python, if there is a variable with the same name ; if yes, Python uses its value.

Otherwise Python would report the error :

```
name < variable > not defined.
```

As you can make out that LEGB rule means checking in the order of Local, Enclosing, Global, Built-in environments (namespaces) to resolve a name reference. (Fig 3.2)

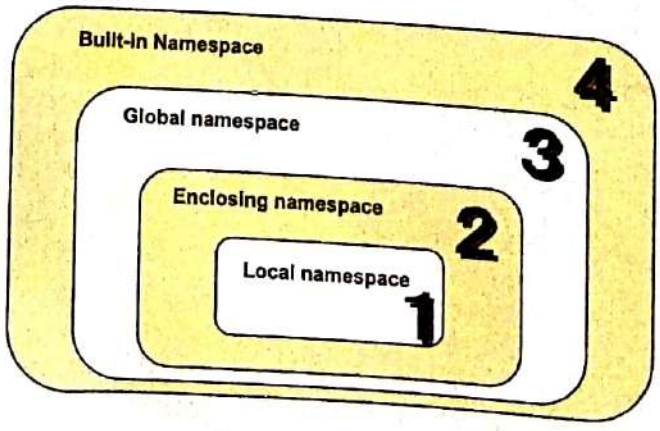



Figure 3.2 LEGB rule for name resolution (scope)

Let us consider some examples to understand this.

## Case 1 : Variable in global scope but not in local scope

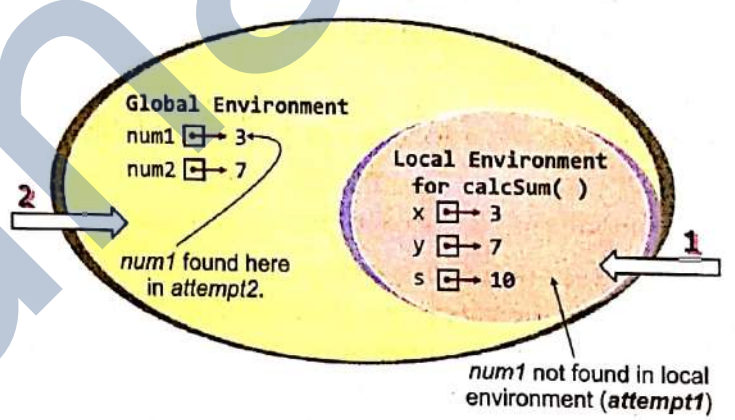
Let us understand this with the help of following code :

```
def calcSum (x, y) :
    s = x + y                    # statement -1-
    print(num1)                  # statement -2-
    return s                     # statement -3-

num1 = int(input( "Enter first number :" ) )
num2 = int(input( "Enter second number :" ) )
print("Sum is", calcSum (num1, num2))
```

*variable **num1** is a global variable, not a local variable*

Consider statement 2 of function ***calcSum( )***. Carefully notice that ***num1*** has not been created in calcSum( ) and still statement2 is trying to print its value. The internal memory status at time of execution of statement 2 of ***calcSum( )*** would be somewhat like :

1. Python will first check the *Local environment* of ***calcSum( )*** for ***num1*** ;
   ***num1*** is not found there.
2. Python now checks for ***num1***, the parent environment of ***calcSum( )***, which is *Global environment* (there is not any intermediate enclosing environment).
   Python finds ***num1*** here ; so it picks its value and prints it.




## Case 2 : Variable neither in local scope nor in global scope

What if the function is using a variable which is neither in its local environment nor in its parent environment ? Simple! Python will return an error, *e.g.*, Python will report error for variable name in the following code as it not defined anywhere :

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

greet( )
```

*This would return error as **name** is neither in local environment nor in global environment*

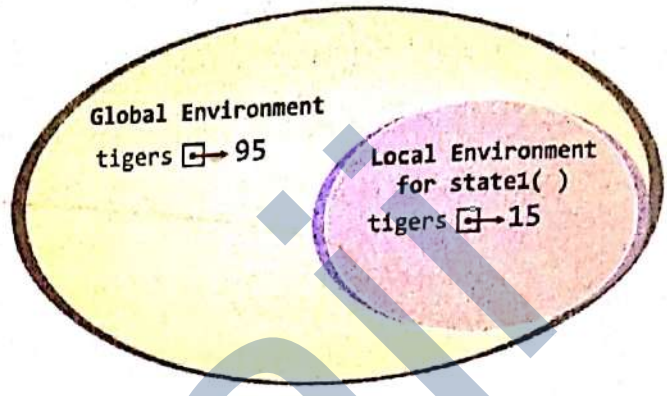
## Case 3 : Same variable name in local scope as well as in global scope

If inside a function, you assign a value to a name which is already there in a higher level scope, Python won't use the higher scope variable because it is an assignment statement and assignment statement creates a variable by default in current environment.

For instance, consider the following code ; read it carefully :

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

tigers = 95
print tigers
state1()
print(tigers)
```

*This statement will create a local variable with name tigers as it is assignment statement. It won't refer to tigers of main program.*




The above program will give output as :

```
95
15
95
```

*Result of print statement inside state1( ) function, thus, value of local **tigers** is printed.*

*Result of print statement inside main program, thus, value of global **tigers** is printed.*

That means a local variable created with same name as that of global variable, it hides the global variable. As in above code, local variable **tigers** hides the global variable **tigers** in function ***state1( )***.

## 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.

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>
```

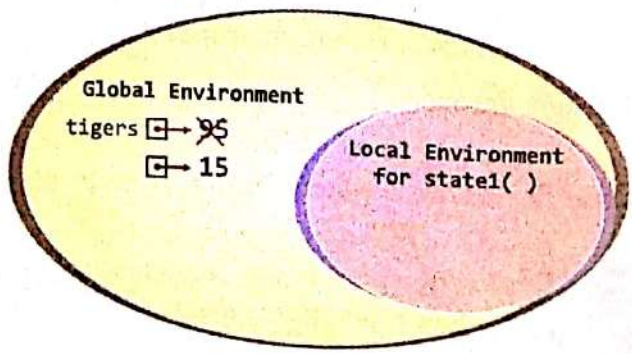
**NOTE**

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*.

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**.*



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.*

**NOTE**

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

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.

**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.

### CALLING FUNCTIONS, ARGUMENT TYPES, SCOPE OF VARIABLES — Progress In Python 3.3

This *'Progress in Python'* session is aimed at these concepts : *calling or invoking functions, different types of arguments, scope of variables* and *Name resolution by Python*.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.3 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

## 3.9 MUTABLE/IMMUTABLE PROPERTIES OF PASSED DATA OBJECTS

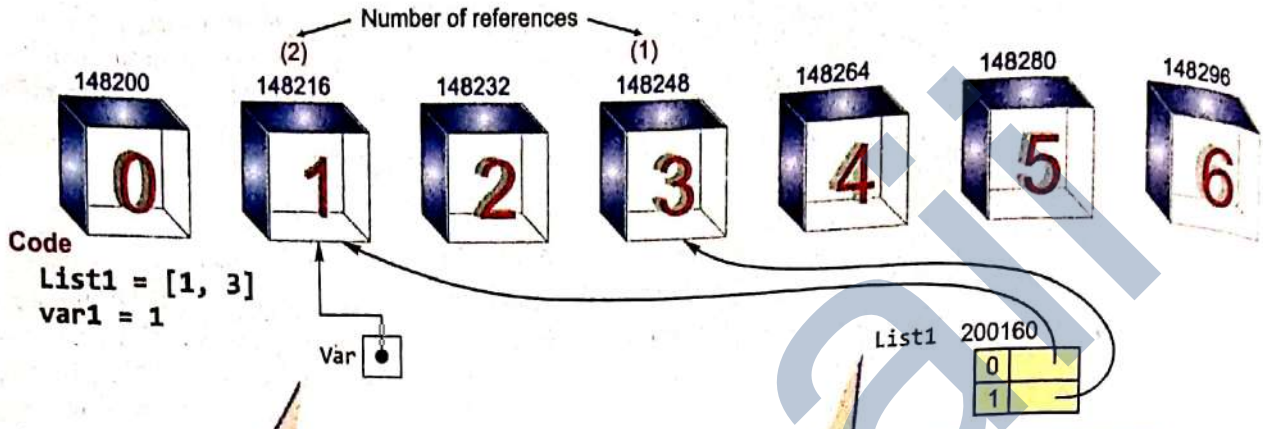
So now you know how you can pass values to a function and how a function returns value(s) to its caller. But here it is important to recall following things about variables :

- Python's variables are not *storage containers*, rather Python variables are like memory references; they refer to the memory address where the value is stored.
- Depending upon the mutability/immutability of its data type, a variable behaves differently. That is, if a variable is referring to an immutable type then any change in its value will also change the memory address it is referring to, but if a variable is referring to mutable type then any change in the value of mutable type will not change the memory address of the variable (recall section 1.7). Following figure also summarizes the same.

**IMPORTANT**

Another important thing to know here is that the *scope rules* apply to the *identifiers* or the *name labels* and not on the values. For example, if you pass an argument ***a*** with value 5 to a function that receives it in parameter ***b***, then both ***a*** and ***b*** will refer to same value (*i.e.*, 5) in data space (even with different scopes, they may refer to same data in data space depending upon mutability), but, you can use name ***b*** only inside the called function and not in the calling function. This will become clearer to you when we discuss mutability with respect to arguments and parameters in the section 3.8.

(Integer literals are stored at some predefined locations)

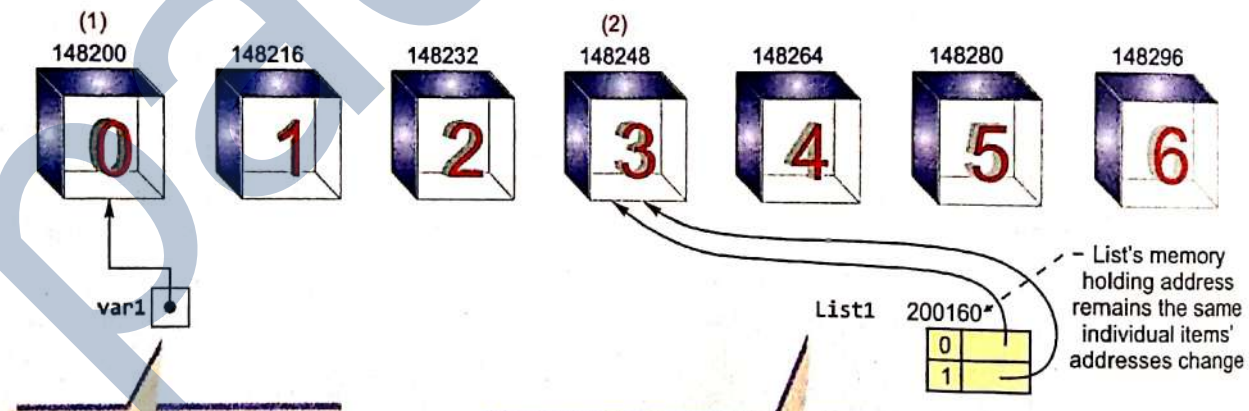



**var1** stores the memory reference of integer value 1 (*i.e.*, 148216) as it is currently storing 1

**List1** has been allocated address 200160 where it can store memory references of its individual items *e.g.*, **List1[0]** currently stores here the memory reference of integer value1 (*i.e.*, 148216) and **List[1]** stores the memory reference of integer value 3 (*i.e.*, 148248)

Now if you make following changes to **List1** and **Var1** :

```
var1 = var1 - 1        (Now var1 holds value 0)
List1[0] = 3           (Now first item of List1 is also 3)
```




**var1** now stores memory address of integer 0 (*i.e.*, 148200)

The memory allocated to List1, where it can hold its data items' memory address **remains the same** *i.e.*, it still is 200160, BUT as the individual items of **List1** have changed so individual items' memory addresses stored here have to reflect this. Thus both **List1[0]** and **List1[1]** are currently storing memory address 148248, the memory address of integer 3 as both List1[0] and List1[1] currently hold integer value 3.

Figure 3.4 Impact of Mutability and Immutability

### 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)
```

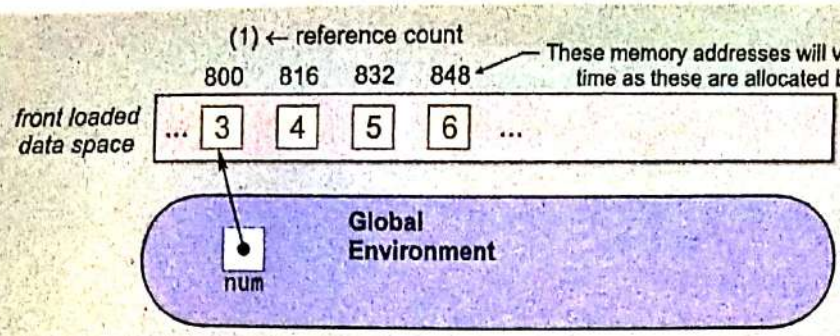
The value got changed from 3 to 8 inside function BUT NOT got reflected to __main__

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.

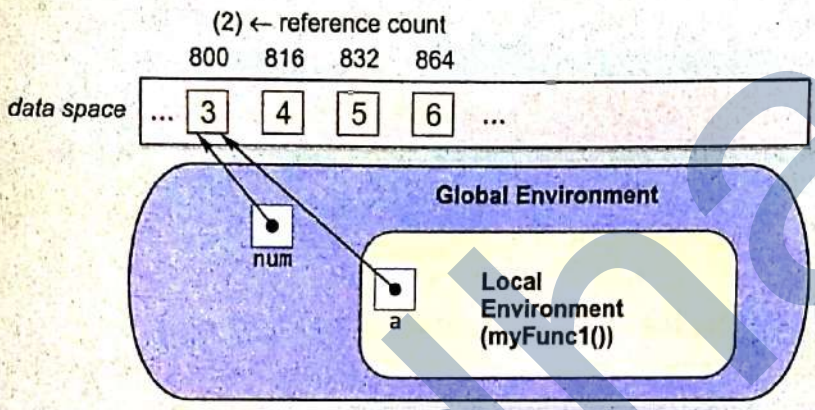
Let us see how the memory environments are created for above code (*i.e.*, ***sample code1***).

## Memory Environment For Sample Code 1.1

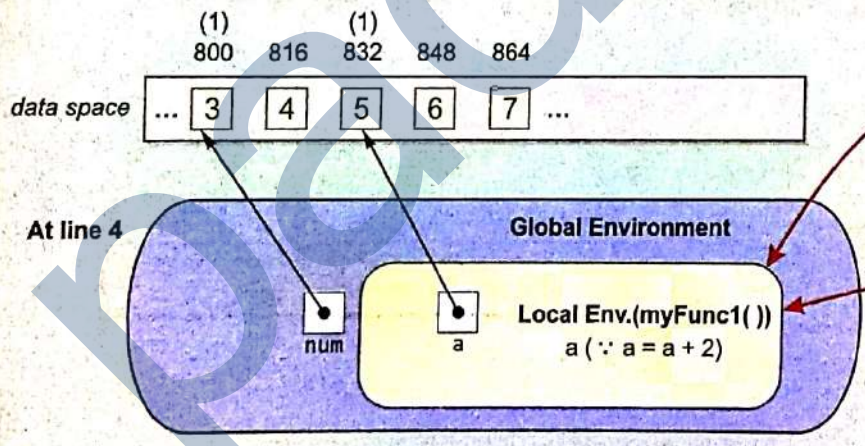
(Note : Passed value is an integer, an immutable type)




*Till Lines 7-9 of code of –main–*

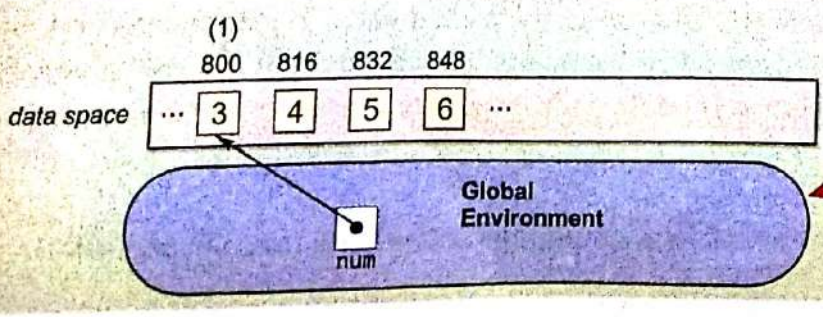



*At **line10** (function is called) argument num is received in parameter **a** and for **lines 1, 2, 3**, environment remains the same*




*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**.*

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__

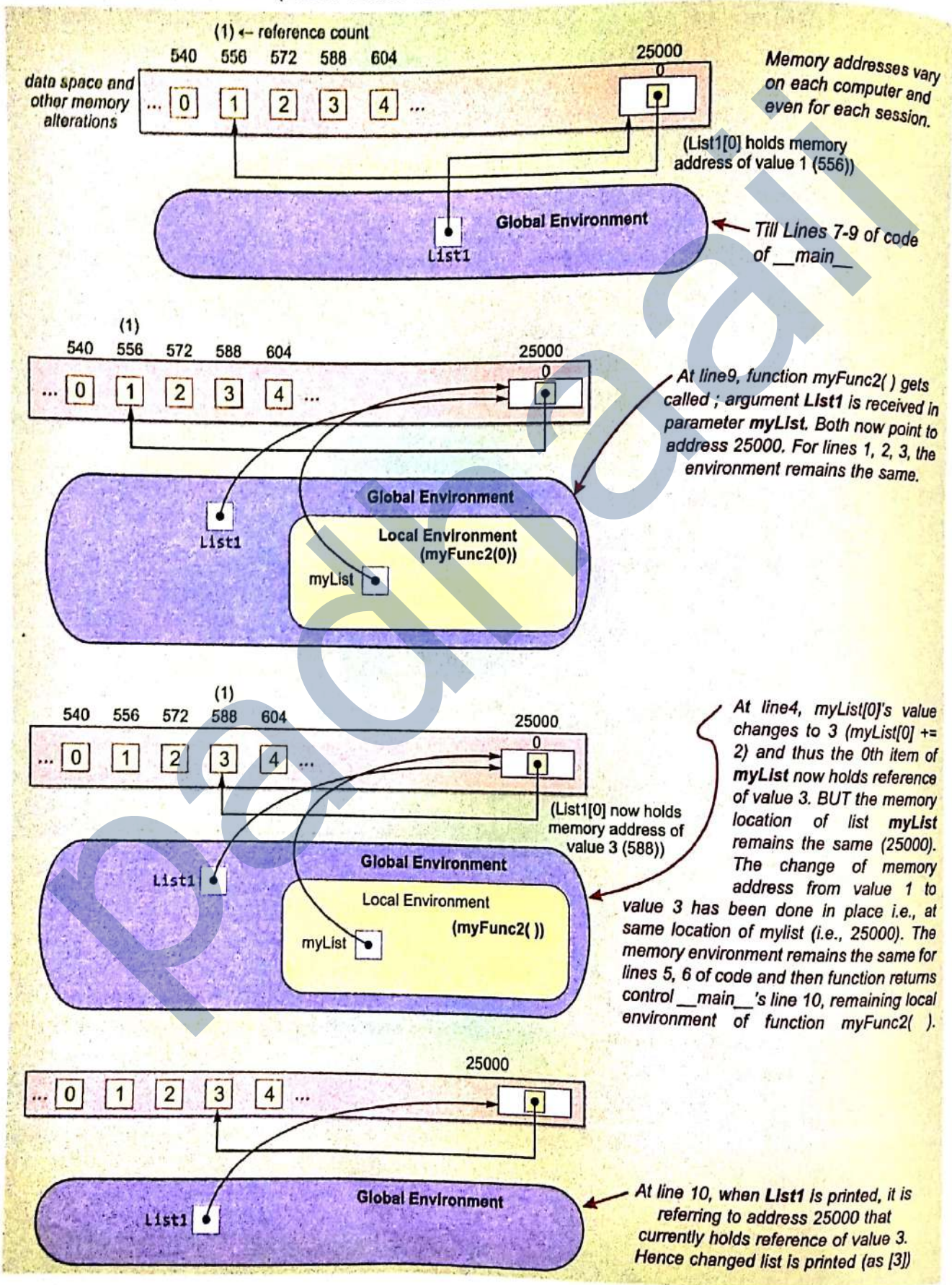
As you can see that the function myFunc2( ) receives a mutable type, ***a list***, this time. The passed list (**List1**) contains value as [1] and is received by the function in parameter **mylist**. The changes made inside the function in the list ***mylist*** get reelected in the original list passed,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

## Memory Environment For Sample Code 2.1

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



So changes in memory references occurred at the same address 25000 for both lists (argument and parameter) and hence got reflected in __main__ , the very essence of mutable types.

Similarly, if you make other changes such as adding or deleting items from a passed list, these will also get reflected back as both the passed list argument and received parameter list work with the same memory address where changes are done **in place**. Check following code and its output – changes are reflected back in __main__

## Sample Code 2.2

***Passing a Mutable Type Value to a function – Adding/Deleting items to it)***

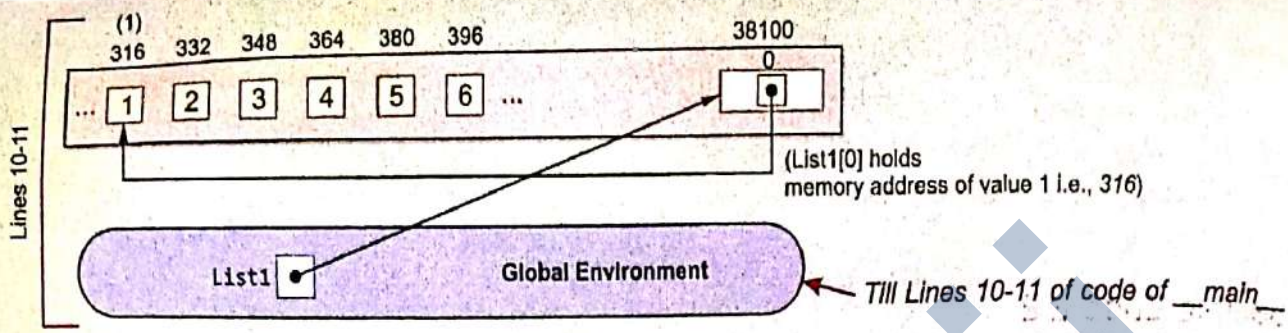
```
1.  def myFunc3(myList):
2.      print("\t Inside CALLED Function now")
3.      print("\t List received:", myList)
4.      myList.append(2)
5.      myList.extend([5,1])
6.      print("\t List after adding some elements:", myList)
7.      myList.remove(5)
8.      print("\t List within called function, after all changes:", myList)
9.      return
10. List1 = [1]
11. print("List before function call :", List1)
12. myFunc3(List1)
13. print("\List 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 after adding some elements: [1, 2, 5, 1]
  List within called function, after all changes: [1, 2, 1]
List after function call :  [1, 2, 1]
```

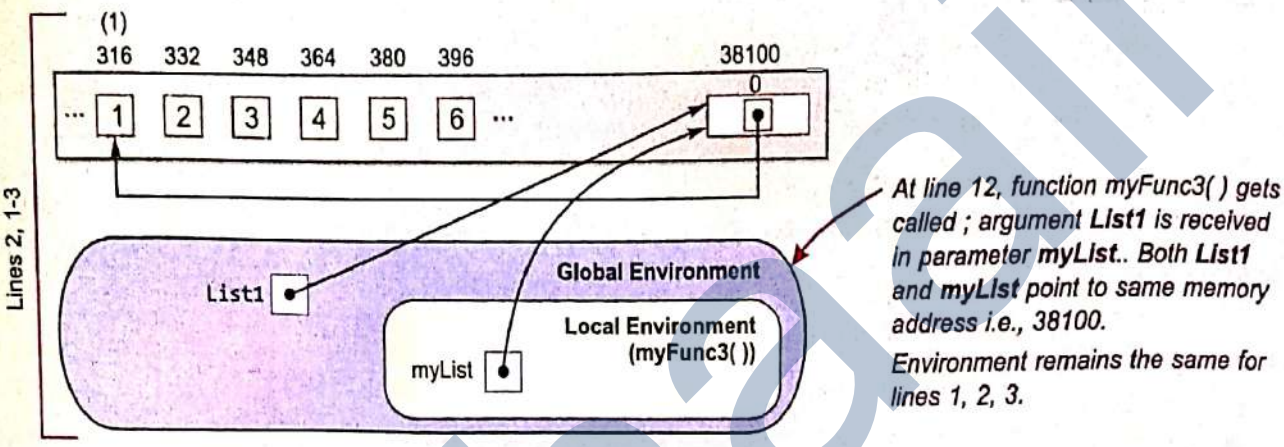
The value got changed from [1] to [1, 2,1] inside function and change GOT REFLECTED to __main__

## Memory Environment For Sample Code 2.2



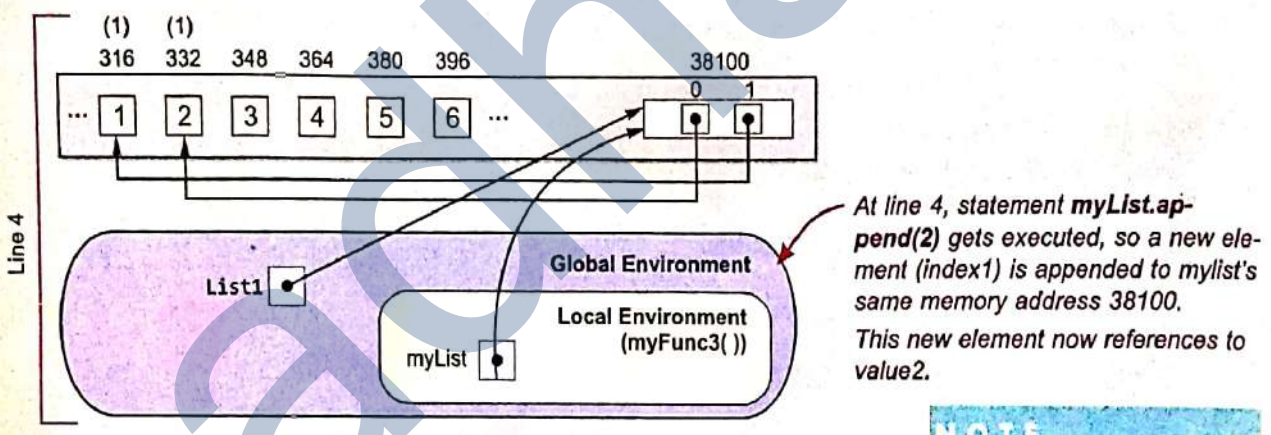

(List1[0] holds memory address of value 1 i.e., *316*)

*Till Lines 10-11 of code of __main__*




*At line 12, function myFunc3( ) gets called ; argument **List1** is received in parameter **myList**.. Both **List1** and **myList** point to same memory address i.e., 38100.*

*Environment remains the same for lines 1, 2, 3.*



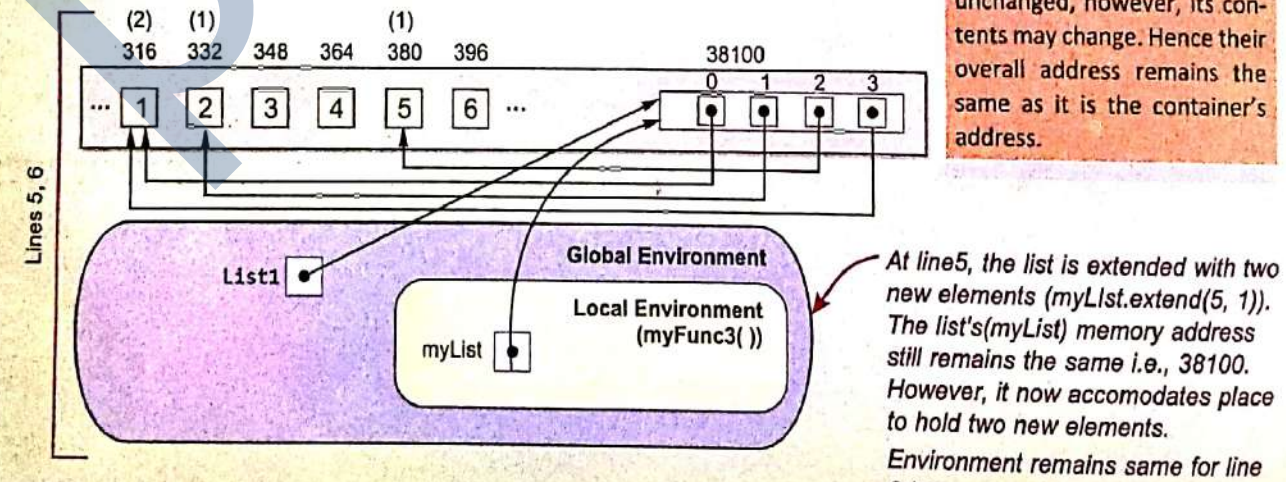

*At line 4, statement **myList.append(2)** gets executed, so a new element (index1) is appended to mylist's same memory address 38100.*

*This new element now references to value2.*

Notice even after appending a new element the list's address is the same *i.e.*, 38100. It, however, now accommodates place to hold new value.

**NOTE**

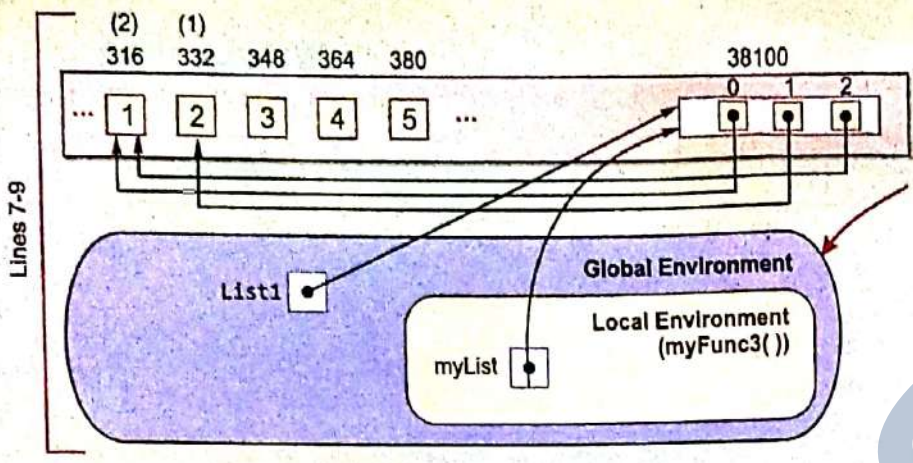
Mutable types act like containers. Container remains unchanged, however, its contents may change. Hence their overall address remains the same as it is the container's address.




*At line5, the list is extended with two new elements (myLIst.extend(5, 1)). The list's(myList) memory address still remains the same i.e., 38100. However, it now accomodates place to hold two new elements.*

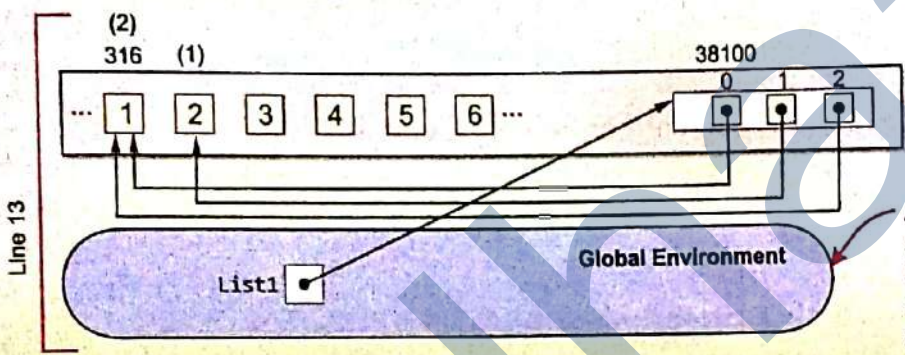
*Environment remains same for line 6 too.*

## Memory Environment For Sample Code 2.2 (Contd....)




*At line 7, statement* ***myList.remove(5)*** *removes element 5 from list and thus only three elements are left in List(**myList**). But the memory address of the list (myList) remains the same even after changes in its contents.*

*The environment remains the same till line 9 and then control returns to line13.*




*At line 13, after returning from myFunc3( ) the local environment is removed. The list. List1 is still referencing the address 38100 with all the changes intact.*

## Sample Code 2.3

***Passing a Mutable Type Value to a function – Assigning parameter to a new value/variable.***

```
1.  def myFunc4(myList):
2.      print("\n\t Inside CALLED Function now")
3.      print("\t List received:", myList)
4.      new = [3,5]
5.      myList = new
6.      myList.append(6)
7.      print("\t List within called function, after changes:", myList)
8.      return

9.  List1 = [1, 4]
10. print("List before function call :", List1)
11. myFunc4(List1)
12. print("\nList after function call :", List1)
```

Parameter list get assigned a new value (a list here)

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

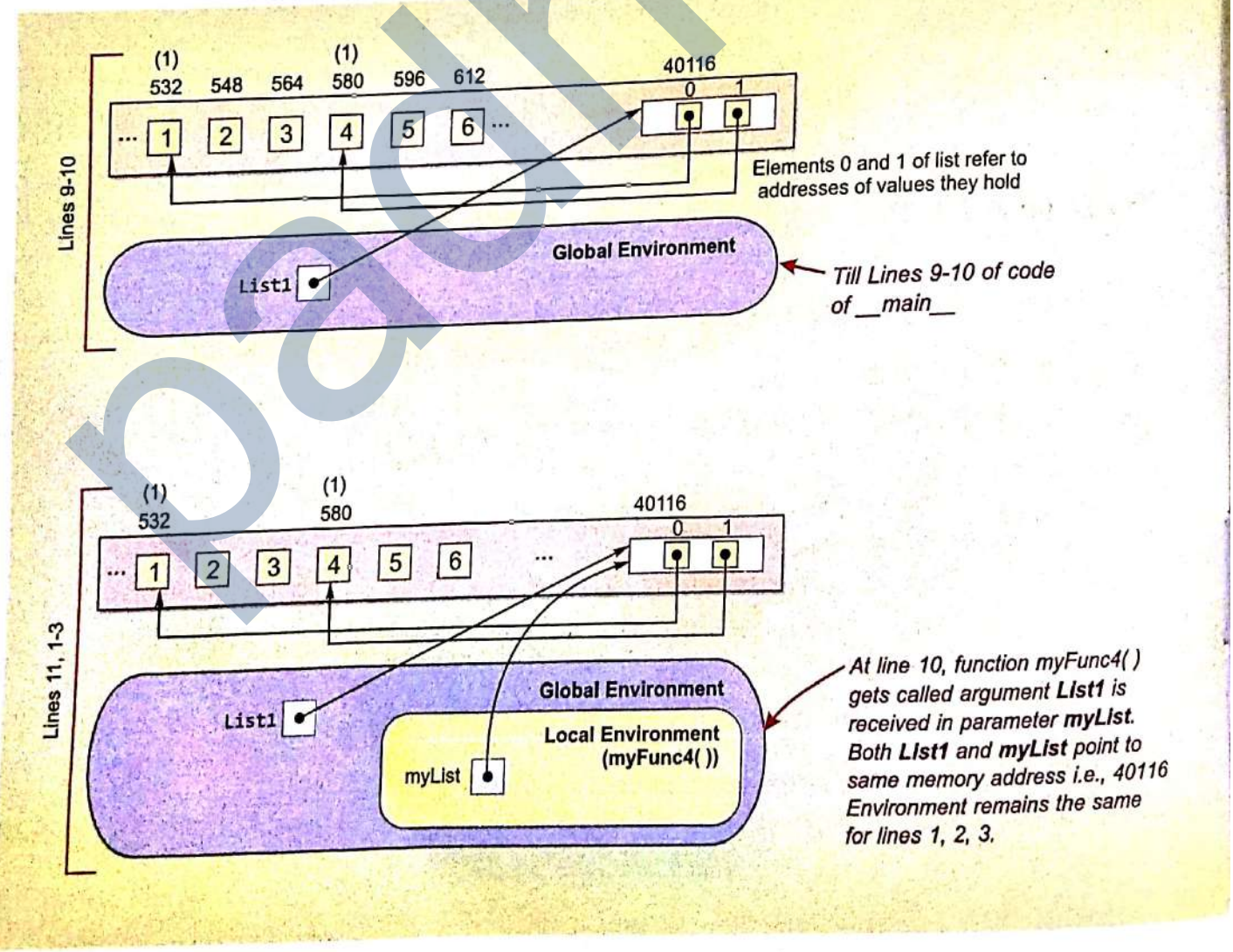
```
List before function call :  [1, 4]

  Inside CALLED Function now
  List received:  [1, 4]
  List within called function, after changes: [3, 5, 6]

List after function call :  [1, 4]
```

The value got changed from [1, 4] to [3,5, 6] inside function and change DID NOT GET REFLECTED to __main__

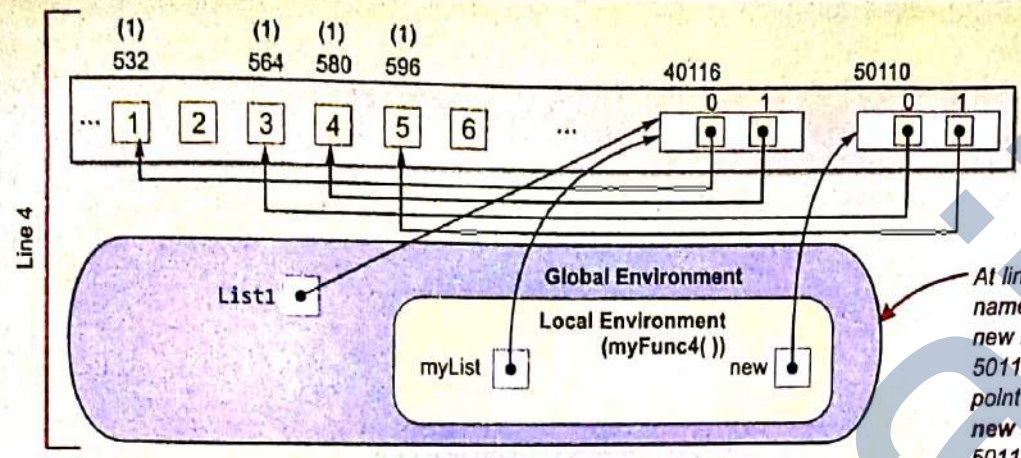
Isn't this unexpected, the passed value is of a mutable type, a list. Right ? But still the changes made in the function do not get reflected back to __main__. Why ? No worries. Let's find out. But first have a look at the memory environments created for above code.

## Memory Environment For Sample Code 2.3

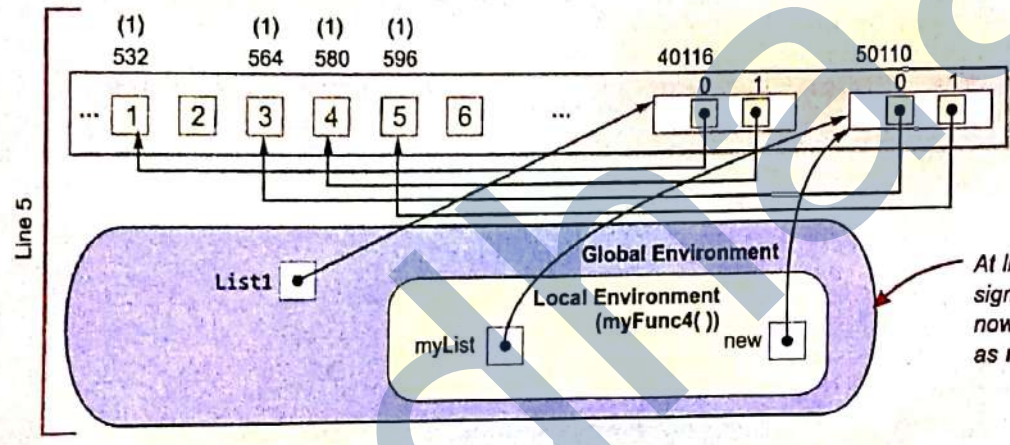
(Note : Passed value is a list, a mutable type)



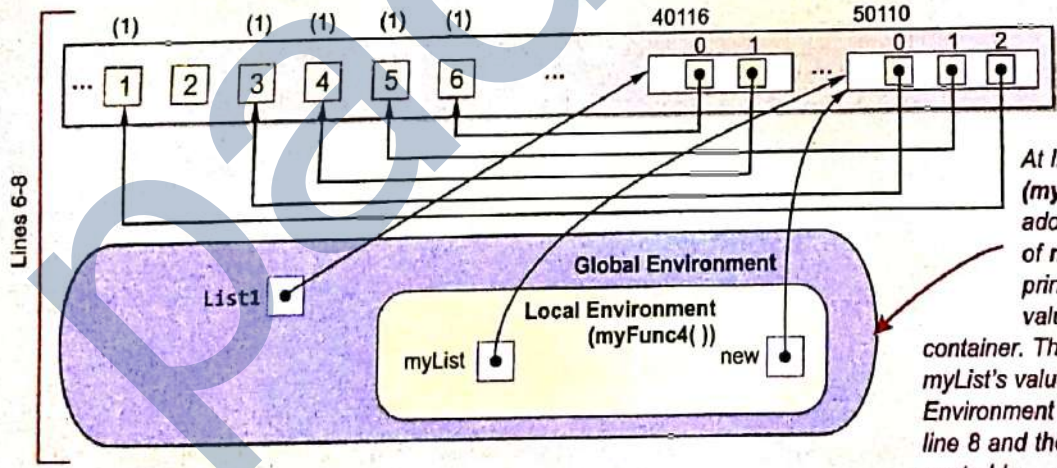
## Memory Environment For Sample Code 2.3 (Contd....)



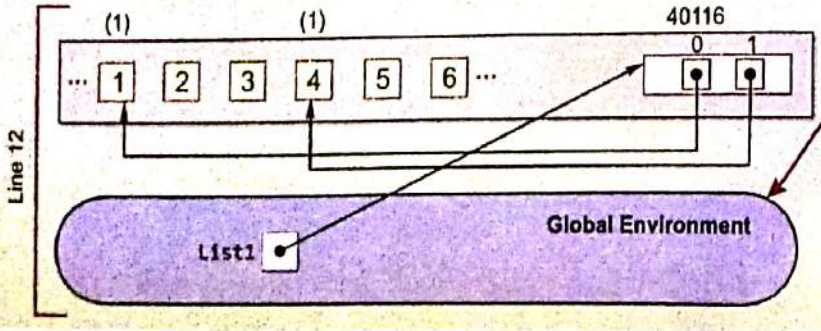

*At line 4, a new list by the name new is created for which new memory is allocated at 50110. While **List1** and **myList** point to memory address 40116, **new** has memory address as 50110.*




*At line 5, **myList** is assigned **new**, which means it now points to same address as **new** i.e., 50110.*




*At line 6, statement **(myList.append(6))** adds value 6 at the end of **myList**. Since **myList** is printing to address 50110, value 6 gets added to this container. Thus when you print myList's value, it shows [3, 5, 6]. Environment remains the same till line 8 and then function returns control to __main__ at line 12.*




*After myFunc4( ) gets over, the local environment is removed. The list, **List1** is still referring to address 40116 and prints whatever is contained at this very address i.e., [1, 4]. Changes that were made to address 50110 are not reflected.*

As it is clear from the memory environs of above code that the moment you make the parameter refer to another address by assigning a different variable name, it loses the address of the actual argument passed to it. So, whatever changes are made to it post this will not get reflected to original argument even though it was mutable because the addresses became different. So we can summarize the mutable behaviour as summarized below :

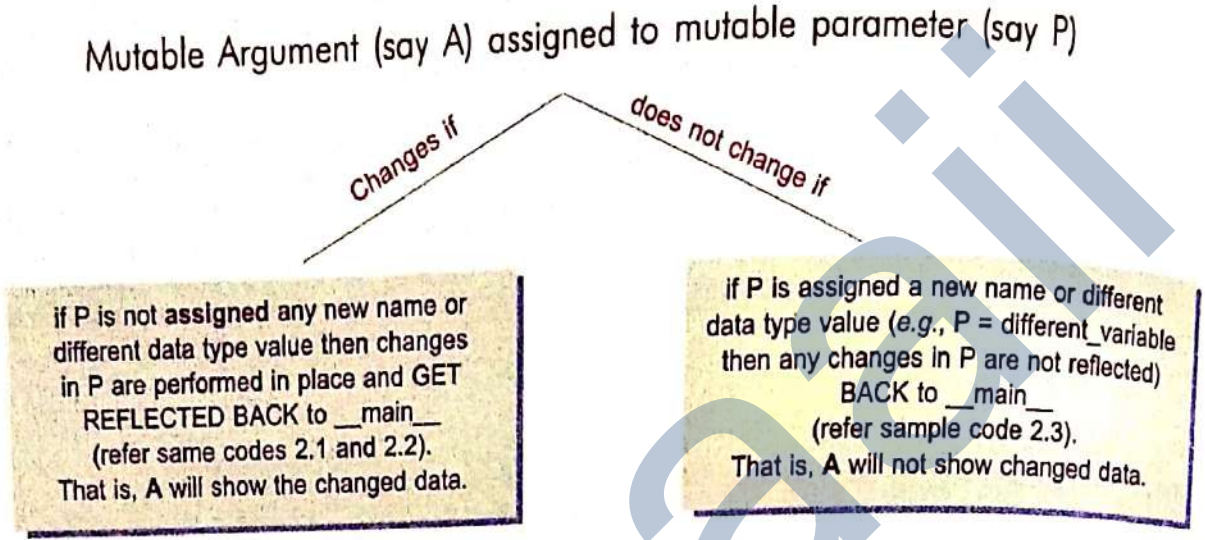



Figure 3.5 Mutable types' behaviour with functions.

That means, we can say that :

- Changes in immutable types are not reflected in the caller function at all.
- Changes, if any, in mutable types
  - are reflected in caller function if its name is not assigned a different variable or datatype.
  - are not reflected in the caller function if it is assigned a different variable or datatype.

**Check Point 3.1**

1. If return statement is not used inside the function, the function will return:
   (a) 0 (b) None object
   (c) an arbitrary integer
   (d) Error! Functions in Python must have a return statement.
2. Which of the following keywords marks the beginning of the function block?
   (a) func (b) define (c) def (d) function
3. What is the area of memory called, which stores the parameters and local variables of a function call ?
   (a) a heap (b) storage area
   (c) a stack (d) an array
4. Find the errors in following function definitions :

(a)
```
def main( )
    print ("hello")
```

(b)
```
def func2() :
    print (2 + 3)
```

(c)
```
def compute( ) :
    print (x * x)
```

(d)
```
square (a)
    return a * a
```

## MUTABILITY/IMMUTABILITY OF ARGUMENTS — Progress In Python 3.4

This 'Progress in Python' session is aimed at these concepts : *Mutability/Immutability of the arguments, parameters and their behaviour in various situations.*

⋮

>>>❖<<<

## 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.

## Solved Problem

1. *What is the significance of having functions in a program ?*

Solution. Creating functions in programs is very useful. It offers following advantages :

(i) *The program is easier to understand.*

Main block of program becomes compact as the code of functions is not part of it, thus is easier to read and understand.

(ii) *Redundant code is at one place, so making changes is easier.*

Instead of writing code again when we need to use it more than once, we can write the code in the form of a function and call it more than once. If we later need to change the code, we change it in one place only. Thus it saves our time also.

(iii) *Reusable functions can be put in a library in modules.*

We can store the reusable functions in the form of modules. These modules can be imported and used when needed in other programs.

2. *From the program code given below, identify the parts mentioned below :*

```
def processNumber(x):
    x = 72
    return x + 3

y = 54
res = processNumber(y)
```

**Identify these parts :** *function header, function call, arguments, parameters, function body, main program*

Solution.

| | | |
|---|---|---|
| *Function header* | : | `def processNumber(x) :` |
| *Function call* | : | `processNumber(y)` |
| *Arguments* | : | `y` |
| *Parameters* | : | `x` |
| *Function body* | : | `x = 72`<br>`return x + 3` |
| *Main program* | : | `y = 54`<br>`res = processNumber(y)` |

3. *Trace the following code and predict output produced by it.*

```
1.  def power(b, p) :
2.      y = b ** p
3.      return y
4.
5.  def calcSquare(x) :
6.      a = power(x, 2)
7.      return a
8.
```

```
9.  n = 5
10. result = calcSquare(n) + power(3, 3)
11. print(result)
```

Solution. Flow of execution for above code will be :

$1 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 10 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11$

The output produced by above code will be :

```
52
```

4. *Trace the flow of execution for following programs :*

(a)
```
1   def power(b, p):
2       r = b ** p
3       return r
4
5   def calcSquare(a):
6       a = power(a, 2)
7       return a
8
9   n = 5
10  result = calcSquare(n)
11  print (result)
```

(b)
```
1. def increment(x) :
2.     x = x + 1
3.
4. # main program
5. x = 3
6. print(x)
7. increment(x)
8. print(x)
```

(c)
```
1.  def increment(x):
2.      z = 45
3.      x = x + 1
4.      return x
5.
6.  # main
7.  y = 3
8.  print(y)
9.  y = increment(y)
10. print(y)
11. q = 77
12. print(q)
13. increment(q)
14. print(q)
15. print(x)
16. print(z)
```

Solution.

(a) $1 \rightarrow 5 \rightarrow 9 \rightarrow 10 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 10 \rightarrow 11$

(b) $1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 1 \rightarrow 2 \rightarrow 8$

*Control didnot return to function call statement (7) as nothing is being returned by increment( )*

(c) $1 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 14 \rightarrow 15 \rightarrow 16$

5. *What is the difference between the formal parameters and actual parameters? What are their alternative names? Also, give a suitable Python code to illustrate both.*

Solution. ***Actual Parameter*** is a parameter, which is used in *function call* statement to send the value from *calling function* to the *called function*. It is also known as ***Argument***.

***Formal Parameter*** is a parameter, which is used in *function header* of the *called function* to receive the value from *actual parameter*. It is also known as ***Parameter***.

*For example,*

```
def addEm(x, y, z):
    print(x + y + z)

addEm(6, 16, 26)
```

In the above code, *actual parameters* are **6, 16** and **26** ; and *formal parameters* are **x, y** and **z**.

6. *Consider a function with following header :*

```
def info(object, spacing = 10, collapse = 1):
```

*Here are some function calls given below. Find out which of these are correct and which of these are incorrect stating reasons :*

```
a.  info( obj1 )
b.  info(spacing = 20)
c.  info( obj2, 12 )
d   info( obj11, object = obj12)
e.  info( obj3, collapse = 0)
f.  info( )
g.  info(collapse = 0, obj3 )
e.  info( spacing = 15, object = obj4 )
```

Solution.

| | | |
|---|---|---|
| (*a*) | Correct | *obj1* is for positional parameter **object** ; **spacing** gets its default value of *10* and **collapse** gets its default value of *1*. |
| (*b*) | Incorrect | Required positional argument (object) missing ; required arguments cannot be missed. |
| (*c*) | Correct | Required parameter **object** gets its value as *obj2* ; **spacing** gets value *12* and for skipped argument **collapse**, default value *1* is taken. |
| (*d*) | Incorrect | Same parameter object is given multiple values – one through positional argument and one through keyword(named) argument. |
| (*e*) | Correct | Required parameter **object** gets its value as *obj3* ; **collapse** gets value *0* and for skipped argument **spacing**, default value *10* is taken. |
| (*f*) | Incorrect | Required parameter **object's** value cannot be skipped. |
| (*g*) | Incorrect | Positional arguments should be before keyword arguments. |
| (*h*) | Correct | Required argument **object** gets its value through a keyword argument. |

7. *What is the default return value for a function that does not return any value explicitly ?*

(*a*) None (*b*) int (*c*) double (*d*) null

Solution. (*a*)

8. *What will following code print ?*

```
def addEm(x, y, z):
    print(x + y + z)

def prod (x, y, z) :
    return x * y * z

a = addEm(6, 16, 26)
b = prod ( 2, 3, 6)
print(a, b)
```

Solution.

```
None 36
```

9. *In the previous question's code, identify the void and non-void functions. The previous code stores the return values of both void and non-void functions in variables. Why did Python not report an error when void functions do not return a value?*

Solution.

Void function : addEm( )

Non-void function : prod( )

In Python, void functions do not return a value; rather they report the absence of returning value by returning None, which is legal empty value in Python. Thus, variable *a* stores **None** and it is not any error.

10. *Consider below given function headers. Identify which of these will cause error and why ?*

(*i*) `def func(a = 1, b):`

(*ii*) `def func(a = 1, b, c = 2):`

(*iii*) `def func(a = 1, b = 1, c = 2):`

(*iv*) `def func(a = 1, b = 1, c = 2, d):`

Solution. Function headers (*i*), (*ii*) and (*iv*) will cause error because **non-default arguments cannot follow default arguments.**

Only function header (*ii*) will not cause any error.

11. *What is the difference between a local variable and a global variable ? Also, give a suitable Python code to illustrate both.*

Solution. The differences between a local variable and global variable are as given below :

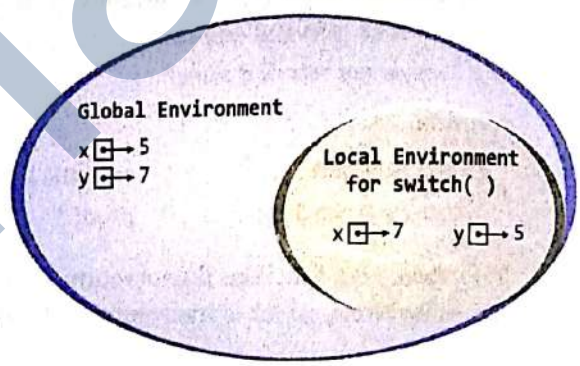
| | Local Variable | Global Variable |
|---|---|---|
| 1. | It is a variable which is declared within a function or within a block | It is variable which is declared outside all the functions |
| 2. | It is accessible only within a function/block in which it is declared | It is accessible throughout the program |

For example, in the following code, *x*, *xCubed* are global variables and *n* and *cn* are local variables.

```
def cube(n):
    cn = n * n * n
    return cn

x = 10
xCubed = cube(x)
print(x,"cubed is", xCubed)
```

12. *Following code intends to swap the values of two variables through a function, but upon running the code, the output shows that the values are swapped inside the switch() function but back again in main program, the variables remain un-swapped. What could be the reason? Suggest a remedy.*

```
def switch(x, y):
    x, y = y, x
    print("inside switch :", end = ' ')
    print("x =", x, "y =", y)

x = 5
y = 7
print("x =", x, "y =", y)
switch(x, y)
print("x =", x, "y =", y)
```




Solution. The reason for un-reflected changes in the main program is that although both main program and *switch( )* have variables with same names *i.e.*, *x* and *y*, but their scopes are different as shown in the adjacent figure.

The scope of *x* and *y* of *switch( )* is local. Though they are swapped in the namespace of *switch( )* but their namespace is removed as soon as control returns to main program. The global variables *x* and *y* remain unchanged as *switch( )* worked with a different copy of values not with original values.

The remedy of above problem is that the *switch( )* gets to work with global variables so that changes are made in the global variables. This can be done with the help of global statement as shown below :

```
def switch(x, y):
    global x, y
    x, y = y, x
    print("inside switch :", end = ' ')
    print("x =", x, "y =", y)

x = 5
y = 7
print("x =", x, "y =", y)
switch (x, y)
print("x =", x, "y=", y)
```

Now the above program will be able to swap the values of variables through switch( ). (Though, now passing parameter is redundant.)

13. *Following code intends to add a given value to global variable a. What will the following code produce?*

```
1. def increase(x) :
2.     a = a + x
3.     return
4.
5. a = 20
6. b = 5
7. increase(b)
8. print(a)
```

Solution. The above code will produce an error.

The reason being whenever we assign something to a variable inside a function, the variable is created as a local variable. Assignment creates a variable in Python.

Thus at line 2, when variable *a* is incremented with the passed value *x*, Python tries to create a local variable *a* by assigning to it the value of expression on the right hand side of assignment. But variable *a* also appears in the right hand side of assignment, which results in error because a is undeclared so far in function.

To assign some value to a global variable from within a function, without creating a local variable with the same name, global statement can be used. So, if we add

```
global a
```

in the first line of function body, the above error will be corrected. Python won't create a local variable *a*, rather will work with global variable *a*.

14. *Which names are local, which are global and which are built-in in the following code fragment?*

```
invaders = 'Big names'
pos = 200
level = 1

def play( ) :
    max_level = level + 10
    print(len (invaders ) == 0)
    return max_level


res = play()
print(res)
```

Solution.

| | | |
|---|---|---|
| **Global names** | : | invaders, pos, level, res |
| **Local names** | : | max_level |
| **Built in** | : | len |

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

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

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

Solution.

```
Python
EasyEasyEasy
```

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

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

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

Solution.

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

17. *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.

18. *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
```

19. *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*.

20. *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
```

21. *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.

22. *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. |

## Assignment

### Type A : Short Answer Questions/Conceptual Questions

1. A program having multiple functions is considered better designed than a program without any functions. Why ?
2. What all information does a function header give you about the function ?
3. What do you understand by flow of execution ?
4. What are arguments ? What are parameters ? How are these two terms different yet related ? Give example.
5. What is the utility of :
   (*i*) default arguments,
   (*ii*) keyword arguments ?
6. Explain with a code example the usage of default arguments and keyword arguments.
7. Describe the different styles of functions in Python using appropriate examples.
8. Differentiate between fruitful functions and non-fruitful functions.
9. Can a function return multiple values ? How ?
10. What is scope ? What is the scope resolving rule of Python ?
11. What is the difference between local and global variables ?
12. When is *global* statement used ? Why is its use not recommended ?
13. Write the term suitable for following descriptions :
    (*a*) A name inside the parentheses of a function header that can receive a value.
    (*b*) An argument passed to a specific parameter using the parameter name.
    (*c*) A value passed to a function parameter.
    (*d*) A value assigned to a parameter name in the function header.
    (*e*) A value assigned to a parameter name in the function call.
    (*f*) A name defined outside all function definitions.
    (*g*) A variable created inside a function body.

## Type B : Application Based Questions

1. What are the errors in following codes ? Correct the code and predict output :

(a)
```
total = 0;
  def sum( arg1, arg2 ):
  total = arg1 + arg2;
  print("Total :", total)
return total;
sum( 10, 20 );
print("Total :", total)
```

(b)
```
def Tot(Number)      #Method to find Total
  Sum = 0
  for C in Range (1, Number + 1) :
    Sum += C
  RETURN Sum
print (Tot[3])       #Function Calls
print (Tot[6])
```

[CBSE D 2015]

2. Consider the following code and write the flow of execution for this. Line numbers have been given for your reference.

```
1    def power(b, p):
2        y = b ** p
3        return y
4
5    def calcSquare(x):
6        a = power(x, 2)
7        return a
8
9    n = 5
10   result = calcSquare(n)
11   print(result)
```

3. What will the following function return ?

```
def addEm(x, y, z):
    print(x + y + z)
```

4. What will the following function print when called ?

```
def addEm(x, y, z):
  return x + y + z
  print(x + y + z)
```

5. What will be the output of following programs ?

(i)
```
num = 1
def myfunc():
    return num
print(num)
print(myfunc())
print(num)
```

(ii)
```
num = 1
def myfunc():
    num = 10
    return num
print(num)
print(myfunc())
print(num)
```

(iii)
```
num = 1
def myfunc():
    global num
    num = 10
    return num
print(num)
print(myfunc())
print(num)
```

(iv)
```
def display():
    print("Hello", end = ' ')
display()
print("there!")
```

6. Predict the output of the following code :

```
a = 10
y = 5
def myfunc():
    y = a
    a = 2
    print("y =", y, "a =", a)
    print("a + y =", a + y)
    return a + y
print("y =", y, "a =", a)
print(myfunc())
print("y =", y, "a =", a)
```

7. What is wrong with the following function definition ?

```
def addEm(x, y, z):
  return x + y + z
  print("the answer is", x + y + z)
```

8. Write a function namely fun that takes no parameters and always returns *None*.
9. Consider the code below and answer the questions that follow :

```
def multiply(number1,number2):
    answer = number1*number2
    print(number1, 'times',number2, '=', answer)
    return(answer)
output = multiply(5,5)
```

(*i*) When the code above is executed, what prints out ?

(*ii*) What is variable output equal to after the code is executed ?

10. Consider the code below and answer the questions that follow :

```
def multiply(number1, number2):
    answer = number1 * number2
    return(answer)
    print(number1,'times', number2, '=',answer )
output = multiply(5,5)
```

(*i*) When the code above is executed, what gets printed ?

(*ii*) What is variable output equal to after the code is executed ?

11. Find the errors in code given below :

(*a*)
```
def minus(total, decrement)
    output = total - decrement
    print(output)
    return (output)
```

(*b*)
```
define check()
    N = input ('Enter N: ')
    i = 3
    answer = 1 + i ** 4 / N
    Return answer
```

(c)
```
def alpha (n, string = 'xyz', k = 10) :
    return beta(string)
    return n

def beta (string)
    return  string == str(n)

print(alpha("Valentine's Day"):)
print(beta (string = ' true '))
print(alpha(n = 5, "Good-bye") :)
```

12. Draw the entire environment, including all user-defined variables at the time line 10 is being executed

```
1.  def sum(a, b, c, d ):
2.      result = 0
3.      result = result + a + b + c + d
4.      return result
5.
6.  def length():
7.      return 4
8.
9.  def mean(a, b, c, d):
10      return float(sum (a, b, c, d))/length()
11.
12. print(sum(a, b, c,d), length(), mean(a, b, c, d))
```

13. Draw flow of execution for above program.
14. In the following code, which variables are in the same scope ?

```
def func1():
  a = 1
  b = 2
def func2():
  c = 3
  d = 4

e = 5
```

15. Write a program with a function that takes an integer and prints the number that follows after it. Call the function with these arguments :

    4, 6, 8, 2 + 1, 4 – 3 * 2, –3 –2

16. Write a program with non-void version of above function and then write flow of execution for both the programs.
17. What is the output of following code fragments ?

(i)
```
def increment(n):
    n.append([4])
    return n
L = [1, 2, 3]
M = increment(L)
print(L, M)
```

(*ii*)
```
def increment(n):
    n.append([49])
    return n[0], n[1], n[2], n[3]
L = [23, 35, 47]
m1, m2, m3, m4 = increment(L)
print(L)
print(m1, m2, m3, m4)
print(L[3] == m4)
```

## Type C : Programming Practice/Knowledge based Questions

1. Write a function that takes amount-in-dollars and dollar-to-rupee conversion price; it then returns the amount converted to rupees. Create the function in both void and non-void forms.
2. Write a function to calculate volume of a box with appropriate default values for its parameters. Your function should have the following input parameters :

   (*a*) length of box ; (*b*) width of box ; (*c*) height of box.

   Test it by writing complete program to invoke it.
3. Write a program to have following functions :

   (*i*) a function that takes a number as argument and calculates cube for it. The function does not return a value. If there is no value passed to the function in function call, the function should calculate cube of 2.

   (*ii*) a function that takes two char arguments and returns True if both the arguments are equal otherwise False.

   Test both these functions by giving appropriate function call statements.
4. Write a function that receives two numbers and generates a random number from that range. Using this function, the main program should be able to print three numbers randomly.
5. Write a function that receives two string arguments and checks whether they are same-length strings (returns True in this case otherwise false).
6. Write a function namely *n*thRoot( ) that receives two parameters $x$ and $n$ and returns $n$th root of $x$ *i.e.*, $x^{\frac{1}{n}}$.

   The default value of $n$ is 2.
7. Write a function that takes a number $n$ and then returns a randomly generated number having exactly $n$ digits (not starting with zero) *e.g.*, if $n$ is 2 then function can randomly return a number 10-99 but 07, 02 etc. are not valid two digit numbers.
8. Write a function that takes two numbers and returns the number that has minimum one's digit.

   [*For example*, if numbers passed are 491 and 278, then the function will return 491 because it has got minimum one's digit out of two given numbers (491's 1 is < 278's 8)].
9. Write a program that generates a series using a function which takes first and last values of the series and then generates four terms that are equidistant *e.g.*, if two numbers passed are 1 and 7 then function returns 1 3 5 7.