Chapter- 3

WORKING WITH FUNCTION

Period-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.

Function:

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

Understanding Functions:

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 $2 \times 2^2 = 2$

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

Now, if we represent above polynomial as somewhat as

$f(x) = 2x^2$

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 the def line, meaning it requires a block.
- The statement indented below the function, defines the functionality of the function.
- 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.

Calling/ Invoking/ Using a Function:

To use a function that has been defined earlier, you need to write a function call statement in Python.

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

calcSomething(5)

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

a = 7

calcSomething(a)

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

Consider one more function definition given below:

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

(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 expression as argument.

```
num = 10
cube (num+5)
```

(v) Using function call inside another statement.

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

(vi) Using function call inside expression.

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

In a nutshell:

1. Parameters can have only variables.
2. Arguments can have.

i. Literals
ii. Variables
iii. Expressions
iv. Combination of all above

Common mistakes to be avoided:

1. **def is a keyword hence must be written in lowercase.(usually students write as DEF/Def)**
2. **The function header must end with a colon symbol (:). (usually skipped)**
3. **Function body must be indented.(indentation is not followed which always causes compilation error)**
4. **A function is defined only once but it can be called n no. of times.(multiple definition of a single function leads to error)**

Period-2

Python Function Types

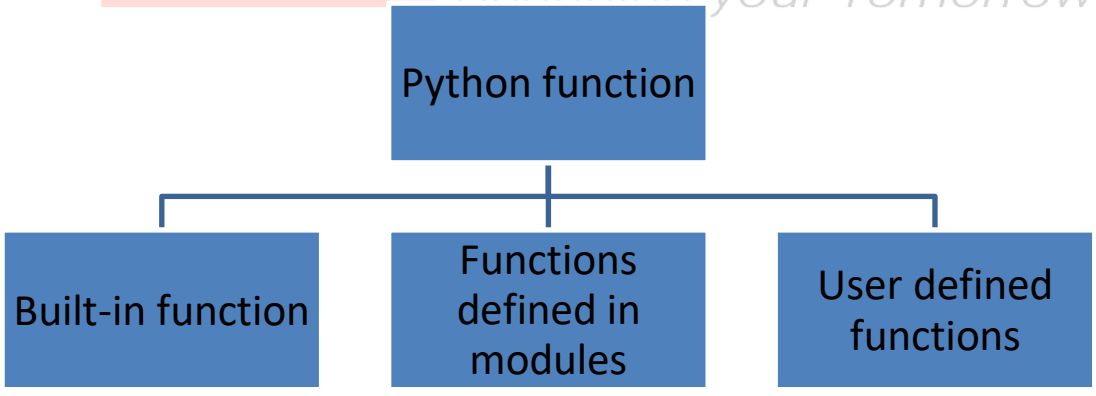
Python comes preloaded with many function-definitions that Python functions can belong to one of the following three categories.

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.

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

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

This chapter will enable you to create your own functions called user defined functions.



Defining Functions in Python:

A function once defined can be invoked as many times as needed by using its name, without having to rewrite its code.

For example, consider some function definitions given below:

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

Or

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

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.

 # Sample Code 1

```
def sum0f3Multiples1( n ) :
    s = n * 1 + n * 2 + n * 3
    return s
```

#Sample Code 2

```
def sum0f3Multiples2( n ) :
    s = n * 1 + n * 2 + n * 3
    print(s)
```

#Sample Code 3

```
def area0fSquare ( a ) :
```

```
        return a * a
```

 # Sample Code 4

```
def perimeterCircle( r ) :
    return (2 * 3.1459 * r)
```

#Sample Code 5

```
def Quote( ) :
    print("\t Quote of the Day")
    print("Act Without Expectations!!")
    print ("\t-Lao Tzu")
```

A function definition defines a user-defined function. It does not execute the function body; this gets executed only when the function is called or invoked.

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. The Python interpreter starts the execution of a program/script from the top-level statement. The top-level statements are part of the main program. Internally Python gives a special name to top level statements as main.

```
def function1( ) :
        :
def function2 ( ) :
        :
def function3( ) :
        :
        :
# top – level statements here
statement1
statement 2
```

:

__name__

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__)

Example:

```
def greet( ) :
    print("Hi there!")
    print("At the top-most level right now")
    print ("Inside", __name__)
```

Upon executing above program, Python will display:

At the top-most level right now

Inside __main__

Common mistakes to be avoided:

1. **While using the functions defined in a module, the respective module should be imported. (students use the functions without importing the respective modules which causes error)**
2. **Execution of the program starts from the top level i.e., the main.(students usually do mistake by executing the code from its beginning)**

Period-3

Flow of Execution in a Function Call

You already know that a function is called 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 ().

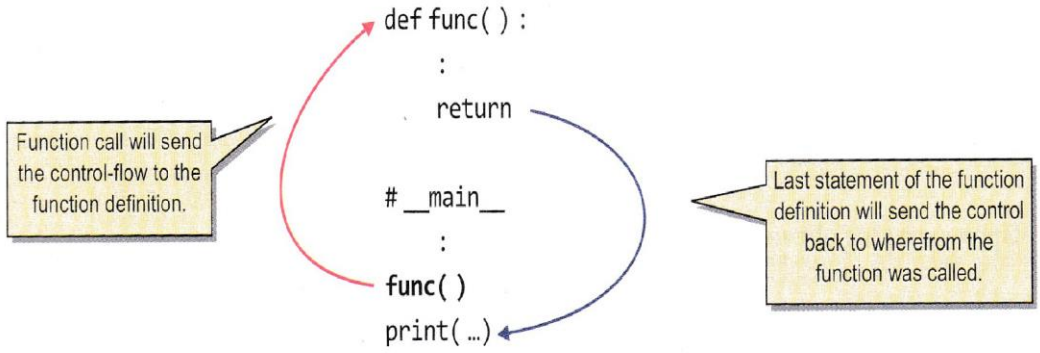
The flow of execution refers to the order in which statements are executed during a program run.

A function body is also a block. In Python, a block is executed in an **execution frame**.

An execution frame contains:

- Some internal information (used for debugging).
- Name of the function.
- Values passed to function.
- Variables created within function.
- Information about the next instruction to be executed.

Whenever a function call statement is encountered, an execution frame is created and the 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:



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)
```

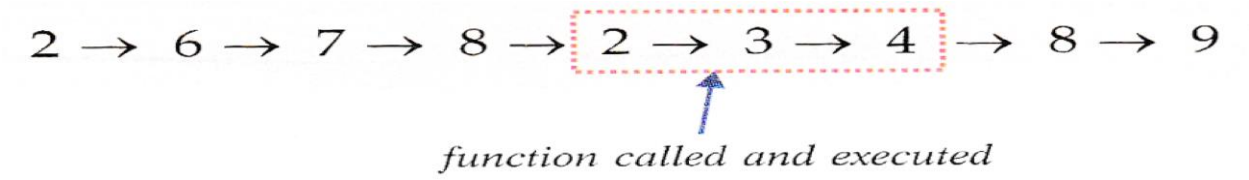
Program execution begins with the first statement of the main segment.

Actual flow of execution:

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. num 2 = 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:



Explanation

- Line 1 is ignored because it is a comment.
- Line 2 is executed and determined that it is a function header, so the 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 the first line of function-body, i.e., line 3, function returns after line 4 to line containing function call statement i.e., line 8 and then to line 9.

Caller & callee

A 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 calSum( ) function.

Common mistakes to be avoided:

1. **While tracing the flow of control of a program if you find a function call statement, the flow must jump to its respective function header, execute the body, and then comes back to the function call. (usually, students keep on executing sequentially the code that leads to wrong execution flow)**

Period-4

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 the above program has a function namely multiply( ) that receives two values. This function is being called thrice passing different values. The three function calls of 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 variable a and b in function header will receive values y and x, respectively.

Let us define these two types of values more formally.

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

So, you can say that arguments appear in function call statements and parameters appear in function header.

Arguments in Python can be one of these value types:

(1)Literals (2) Variable (3) Expressions

But the parameters in Python must 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**.

Thus, for a function as defined below:

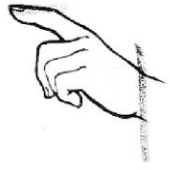
```
def multiply ( a, b) :
    print ( a * b )
```

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.

If you are passing values of immutable types (e.g., number, 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 of dictionaries) then called function would be able to make changes in them.

Common mistakes to be avoided:

1. **Arguments are the values passed in the function call which are also referred to as actual argument or actual parameter. Parameters are passed in the function header which are also known as formal parameters or formal argument. (students usually intermix these terms)**

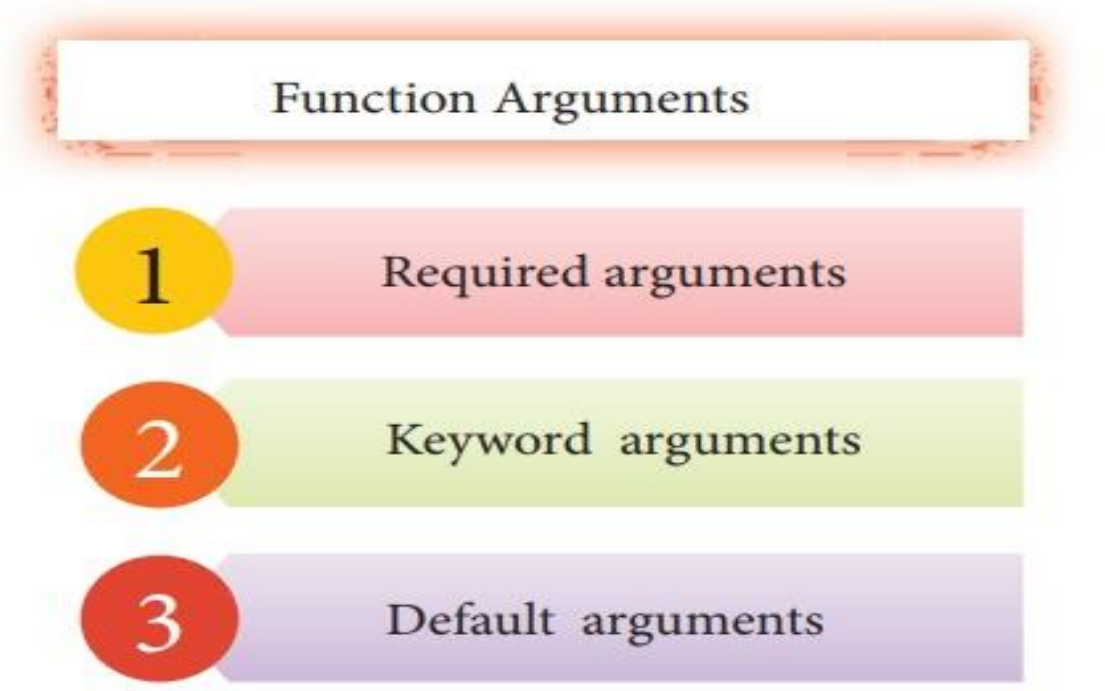
Period-5

Passing Parameters:

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.



Positional/Required Arguments:

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.
```

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 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.

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%, and 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.

def interest (principal, time, rate = 0.10)

This is the 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 above function declaration provides a default value of 0.10 to the parameter rate. Now, if any function call appears as follows:

si_int = interest (5400, 2)

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 (6400,3,0.15)

Then the parameter principal gets value 6400, time gets 3 and the parameter rate gets value 0.15.

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

Period-6

Fact about default argument:

One 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.

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.

Keyword (Named) 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.

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 the above statement is representing a positional argument as it will be assigned to the first parameter based on 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 statements:

- 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.

For instance, consider the following function header:

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

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

Program to calculate simple interest:

```
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)
```

Common mistakes to be avoided:

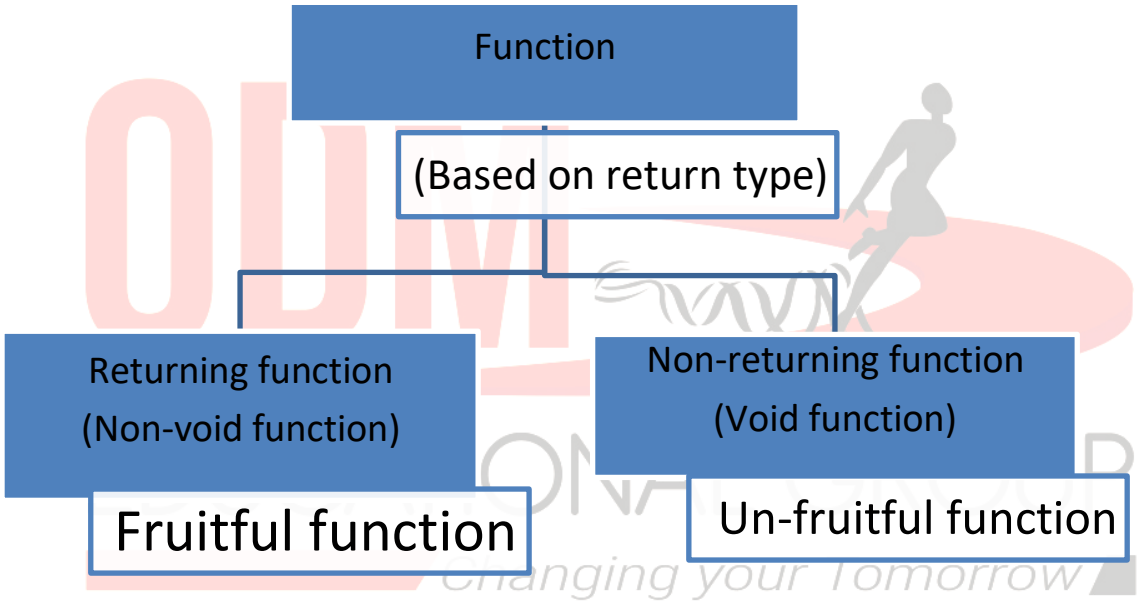
1. **Default arguments should be placed right to left. (usually, students place them in beginning or middle without making the right most argument as default)**
2. **An argument list must first contain positional (required) arguments followed by any keyword argument. (usually, students place keyword argument followed by positional argument)**

Period-7

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 :(based on return statement)

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



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:

(1)Literal **(2) A variable** **(3) An expression**

For examples, following are some legal return statements:

returns 5 # literal being returned.

returns 6 + 4 # expression involving literals being returned.

return a # variable being returned.

return a3** # expression involving a variable and literal, being returned.

return (a+82) / b** # expression involving variables and literals, being returned.

return a + b / c # expression involving variables being returned.

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)

After the function call to sum ( ) function is successfully completed, the returned value will internally substitute the function call statement.

The returned value of a function should be used in the caller function/program inside an expression or a statement.

add_result = sum (a, b) the returned value is being used in the assignment statement.

print(sum(3,4)) The returned value is being used in the print statement.

sum (4, 5) > 6 The returned value is being used in a relational expression.

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)
```

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

check (-15)

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.

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

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

The above program will give output as:

```
helloz
None
```

Consider the following example:

```
# Code 1                    # Code 2
def replicate( ) :          def replicate( ) :
    print("$$$$$")              return "$$$$$"
print(replicate( ) )        print (replicate( ) )
```

Here the output produced by above two codes will be:

Outputs: **Code 1** **Code 2**

$$$$$** **$$$$$

None

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

Returning Multiple Values:

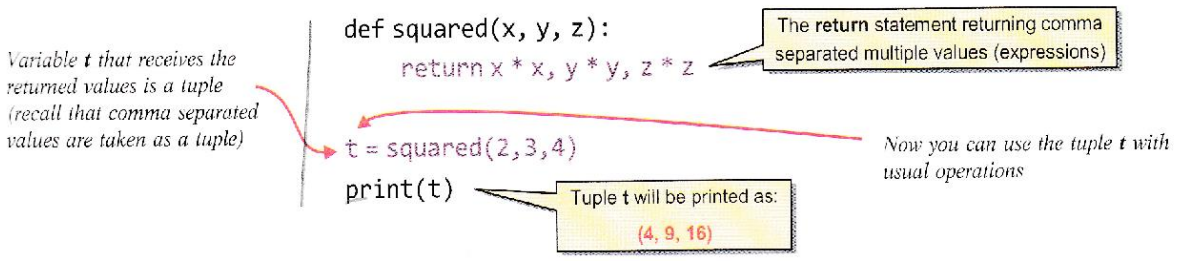
Python lets you return more than one value from a function.

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

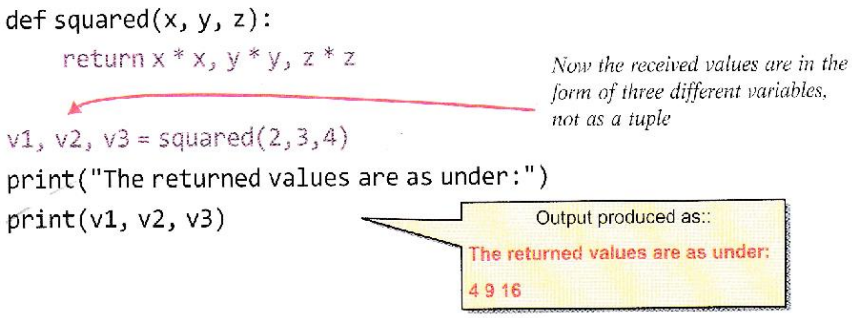
return<value1/variable1/expression1>, <value 2/variable2/expression2>,

(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 that receives two number 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, multi, 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)
```

Common mistakes to be avoided:

1. **Every python function returns a value. If the return statement returns a value, then that value is returned. And if the function does not have a return statement still it returns a None value. (Students think that the function that does not have a return statement returns nothing.)**
2. **When a function returns multiple values and is assigned with a single variable, then the type of the variable becomes a tuple. (students take at as integer)**

Period-8

Composition:

Composition in general refers to using an expression as part of a larger expression or a statement at a part of a 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) Function call as part of larger function call i.e., composition.

  int(str(52) + str(10))

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.

Global Scope:

A name declared in the 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.

Local Scope:

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

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 –
2. num1 = int( input( “Enter first number : “) )          # statement – 1
3. num2 = int( input( “Enter second number : “) )         # statement – 2
4. sum = calcSum (num1, num2 )                           # statement – 3
5. print (‘Sum of  numbers is’, sum)                      # statement – 4
```

A careful look at the program tells that there are three variables mum1, 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( )).

Variables defined outside all functions are global variables.

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

Consider the following example:

```
x = 5
  def func (a) :   ---------- Variable x defined above all functions.
      b = a + 1              It is also a global variable along with y and z.
      return b
  y = input (“Enter number”)
  z = y + func( x ) :
  print (z)
```

Scope Example 2

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

```
1.  def calcSum(a, b, c) :                          # statement – 1 –
2.          s = a + b + c                           # statement – 2 –
3.          return s
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.  num 3 = int (input( “ Number 3 : “)             # statement – 3 –
10. print (“Average of these number is “, average (num1, num2, num3) )
                                                    # statement -4-
```

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 and for local variables, lifetime is their function’s run.

Period-9

Name Resolution (Resolving Scope of a Name):

When you access a variable from within a program or function, Python follows name resolution rule, also known 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.

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 statement 2 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 num 1 :
   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: Some variable name in local scope as well as in global scope

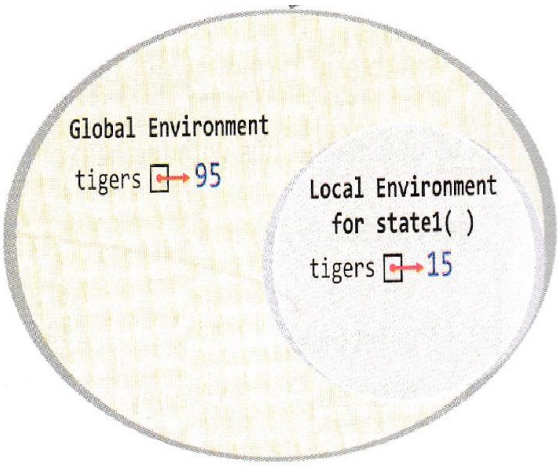
If inside a function, you assign a value to a name which is already there in a higher-level scope. Python will not use the higher scope variable because it is an assignment statement and assignment statement create a variable by default in current environment.

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.

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

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

Case 4: Some variable name in local scope as well as in global scope

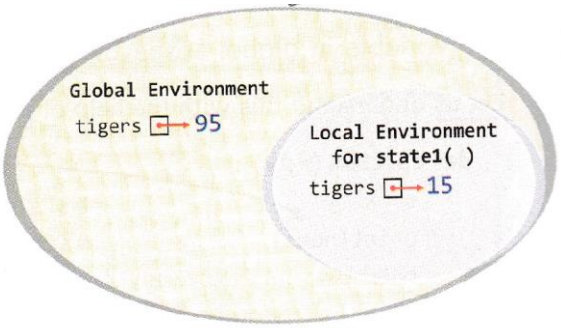
If inside a function, you assign a value to a name which is already there in a higher-level scope. Python will not use the higher scope variable because it is an assignment statement and assignment statement create a variable by default in current environment.

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 hide the global variable tigers in function state1(). What if you want to use the global 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.

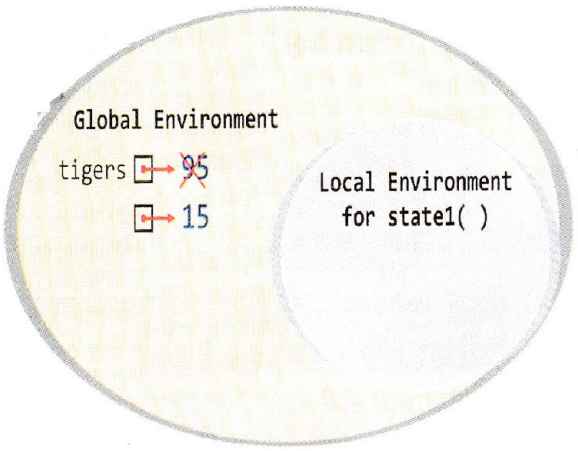
But if you want to assign some value to the global variable without creating any local variable, then what to do? This is because, if you assign any value to a name, Python will create a local variable by the same name. For this kind of problem, Python makes available global statement.

global < variable name>

```
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, that value of global tigers (which is 15 now) is printed.

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

Period 10

Mutable/Immutable Properties of Passed Data Objects

- Python's variables are not storage containers, rater 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. Following figure also summarizes the same.

Mutability/Immutability of Arguments/Parameters and Function Colls

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

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 + 5
5.     print("\t Value of 'a' now charges 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 charges 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.

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]

The value got changed from 1 to 3 inside function and change GOT REFLECTED to – main-.

List after function call : [3]

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__.

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 cal1:", 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 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--

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)
```

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]

Memory Environment For Sample Code 2.3

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