Function

Function are subprograms which are used to compute a value or perform a task.

Type of Functions:-

- Built-in Function

  Ex: - print( ), upper( ) etc

- User-defined Function

Advantage of Function

- Write once and use it as many time as you need. This provides code reusability.
- Function facilitates ease of code maintenance.
- Divide Large task into many small task so it will help you to debug code
- You can remove or add new feature to a function anytime.

Function Definition

We can define a function using def keyword followed by function name with parentheses.

This is also called as Creating a Function, Writing a Function, Defining a Function.

Syntax : -

```
def Function_name ( ):
        Local Variable
        block of statement
        return (variable or expression)
```

Function Body

Syntax : -

```
def Function_name (para1, para2, …):
        Local Variable
        block of statement
        return (variable or expression)
```

Function Body

Note – Need to maintain proper indentation

def represents starting of function definition

parentheses describes this is a function not variable or other object

: describes beginning of function body

def add () **:**

Name of Function

x = 10

y = 20

Local Variable

c = x + y

print(c)

Statements

Function Body

def represents starting of function definition

parentheses describes this is a function not variable or other object

Parameter

: describes beginning of function body

```
def add (y) :
    x = 10
    c = x + y
    print(c)
```

Name of Function

Local Variable

Statements

Function Body

Calling a function

A Function runs only when we call it, function can not run on its own.

Syntax:-

function_name ( )

function_name (arg1, arg2, …)

Ex: -

add ( )

add (20)

add(10.56)

item(“Good Shows”)

```
a = 10
add(a)
```

How Function Work

```
def add () :
        x = 10
        y = 20
        c = x + y
        print(c)
add()
```

```
def add (y) :
        x = 10
        c = x + y
        print(c)
add(20)
```

The parameter *y* do not know which type of value they are about to receive till the value is passed at the time of calling the function. It means the type of data is determined only during runtime not at compile time this is called Dynamic Typing.

Return Statement

Return statements can be used to return something from the function. In Python, it is possible to return one or more variables/values.

Syntax : -

return (variable or expression);

Ex: -

```
return 50
return (50)
return (x + y)
return (y)
return (2, 4)
return (x, y)
```

```
def add (y) :
    x = 10
    c = x + y
    return c
sum = add (20)
print(sum)
```

```
def add (y) :
    x = 10
    return x + y
sum = add (20)
print(sum)
```

```
def add_sub (y) :
    x = 10
    c = x + y
    d = y - x
    return c, d
sum, sub = add (20)
print(sum)
print(sub)
```

Nested Function

When we define one function inside another function, it is known as Nested Function or Function Nesting.

Ex:-

```
def disp():
    def show():
        print("Show Function")
    print("Disp Function")
    show()

disp( )
```

Pass a Function as Parameter

We can pass a function as parameter to another function.

Ex:-

```
def disp(sh):
        print("Disp Function" + sh())

def show():
        return " Show Function"

disp(show)
```

Function return another Function

A function can return another function.

Ex:-

```
def disp():
    def show():
        return "Show Function"
    print("Disp Function")
    return show

r_sh = disp()
print(r_sh())
```

```
def disp(sh):
        print("Disp Function")
        return sh

def show():
        return "Show Function"

r_sh = disp(show)
print(r_sh())
```

Actual and Formal Argument

- Formal Argument - Function definition parameters are called as formal arguments
- Actual Argument - Function call arguments are actual arguments

Formal Arguments

```
def add (x, y) :
    c = x + y
    print(c)

add(10, 20)
```

Actual Arguments

Type of Actual Arguments

- Positional Arguments
- Keyword Arguments
- Default Arguments
- Variable Length Arguments
- Keyword Variable Length Arguments

Positional Arguments

These arguments are passed to the function in correct positional order.

The number of arguments and their positions in the function definition should be equal to the number and position of the argument in the function call.

```
def pw (x, y) :
    z = x**y
    print(z)

pw(5, 2)
```

```
def pw (x, y) :
    z = x**y
    print(z)

pw(2, 5)
```

```
def pw (x, y) :
    z = x**y
    print(z)

pw(5, 2, 3)
```

Keyword Arguments

These arguments are passed to the function with name-value pair so keyword arguments can identify the formal argument by their names.

The keyword argument's name and formal argument's name must match.

```
def show (name, age) :
    print(name, age)

show(name="Geekyshows", age=62)
```

```
def show (name, age) :
    print(name, age)

show(age=62, name="Geekyshows")
```

```
def show (name, age) :
    print(name, age)

show(name="Geekyshows", age=62, roll=101)
```

Note - Number of argument must be equal in formal and actual argument, Not more Not less

Default Arguments

Sometime we mention default value to the formal argument in function definition and we may not required to provide actual argument, In this case default argument will be used by formal argument.

If we do not provide actual argument for formal argument explicitly while calling the function then formal argument will use default value on the other hand if we provide actual argument then it will use provided value

```
def show (name, age=27) :
    print(name, age)

show(name="Good shows")
```

```
def show (name, age=27) :
    print(name, age)

show(name="Good shows", age=62)
```

Note - Number of argument must be equal in formal and actual argument, Not more Not less

Variable Length Arguments

Variable length argument is an argument that can accept any number of values.

The variable length argument is written with * symbol.

It stores all the value in a tuple.

```
def add (*num) :
    z = num[0]+num[1]+num[2]
    print(z)

add(5, 2, 4)
```

```
def add (x, *num) :
    z = x+num[0]+num[1]
    print(z)

add(5, 2, 4)
```

Keyword Variable Length Arguments

Keyword Variable length argument is an argument that can accept any number of values provided in the form of key-value pair.

The keyword variable length argument is written with ** symbol.

It stores all the value in a dictionary in the form of key-value pair.

```
def add (**num) :
    z = num['a']+num['b']+num['c']
    print(z)

add(a=5, b=2, c=4)
```

```
def add (x, **num) :
    z = x+num['a']+num['b']
    print(z)

add(3, a=5, b=2)
```

Local Variables

The variable which are declared inside a function called as Local Variable.

Local variable scope is limited only to that function where it is created. It means local variable value is available only in that function not outside of that function.

```
def add (y) :
    x = 10
    print(x)
    print(x + y)
add(20)
print(x)
```

Local Variable

Using Local variable inside Function

Using Local Variable outside function, it will show error

Global Variables

When a variable is declared above a function, it becomes global variable. These variables are available to all the function which are written after it.

The scope of global variable is the entire program body written below it.

```
a = 50
def show () :
    x = 10
    print(a)
    print(x)

show()
print("x:", x)
print("a:", a)
```

Global Variable

Local Variable

Using Global variable inside Function

Using Local variable inside Function

Using Local Variable outside function, it will show error

Using Global variable

Global Keyword

If local variable and global variable has same name then the function by default refers to the local variable and ignores the global variable.

It means global variable is not accessible inside the function but possible to access outside of function.

In this situation, If we need to access global variable inside the function we can access it using global keyword followed by variable name.

Global Keyword

```
a = 50
def show () :
    a = 10
    print(a)
show()
print(“a:”, a)
```

```
a = 50
def show () :
    global a
    print(a)
    a = 20
    print(a)
show()
print(“a:”, a)
```

globals ( ) Function

This function returns a table of current global variables in the form of dictionary.

```
a = 50
def show () :
    a = 10
    print("Local Variable A:", a)
    x = globals()['a']
    print("X:", x)
show()
print("Global Variable A:", a)
```

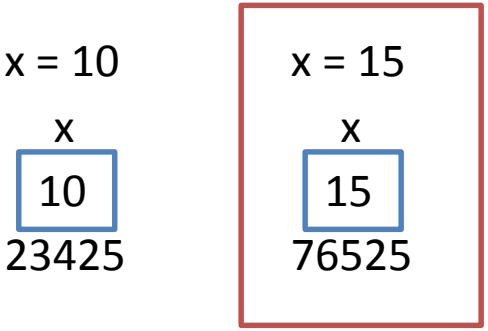
Pass/Call by Object Reference

In C, Java and some other languages we pass value to a function either by value or by reference widely known as “Pass by Value” and “Pass by Reference”.

In Python, Neither of these two concepts is applicable rather the values are sent to functions by means of object reference.

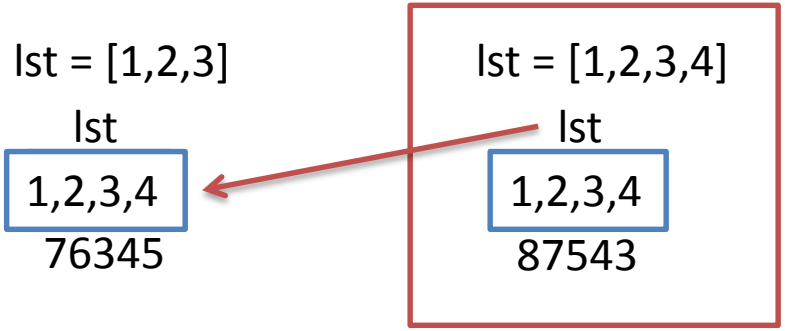
When we pass value like number, strings, tuples or lists to function, the references of these objects are passed to function.

```
def val(x):
    x = 15
    print(x, id(x))

x = 10
val(x)
print(x, id(x))
```



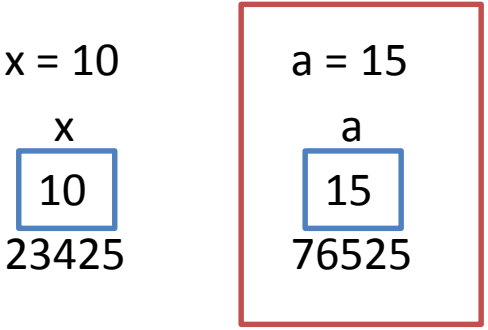
A new object is created in the memory because integer objects are immutable (not modifiable).

```
def val(lst):
    lst.append(4)
    print(lst, id(lst))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```



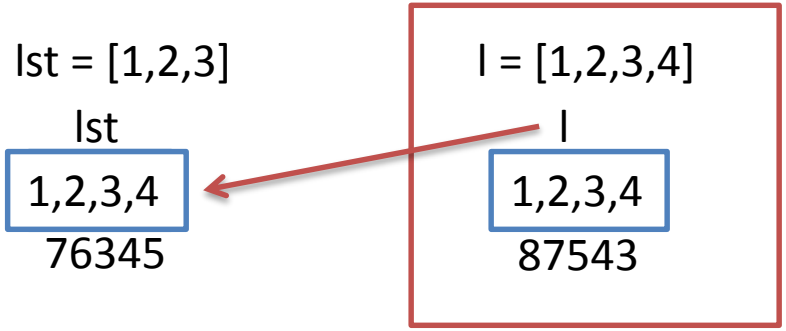
A new object is not created in the memory because list objects are mutable (modifiable). It simply add new element to the same object.

```
def val(a):
    a = 15
    print(a, id(a))

x = 10
val(x)
print(x, id(x))
```



A new object is created in the memory because integer objects are immutable (not modifiable).

```
def val(l):
    l.append(4)
    print(l, id(l))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```



A new object is not created in the memory because list objects are mutable (modifiable). It simply add new element to the same object.

Pass/Call by Object Reference

In Python, values are passed to functions by object references.

If object is immutable (not modifiable) then the modified value is not available outside the function.

If object is mutable (modifiable) then the modified value is available outside the function.

Immutable Objects – Integer, Float, String and Tuple

Mutable Objects – List and Dictionary

```
def val(lst):
    print(lst, id(lst))
    lst = [11, 22, 33]
    print(lst, id(lst))

lst = [1, 2, 3]
print(lst, id(lst))
val(lst)
print(lst, id(lst))
```

lst = [1,2,3]

lst

1,2,3

76345

lst = [11,22,33]

lst

11,22,33

87543

When we create a new object inside function then it will not be available outside function

Recursion

A function calling itself again and again to compute a value is referred to Recursive Function or Recursion.

Expression vs Statement

Expression/Expression Statements

Expression statements are used to compute and write a value, or to call a procedure.

Ex:- Operators like Addition, Subtraction, Function Call etc

Statement

Statements on the other hand, are everything that can make up a line or several lines of Python code. Expressions are also statements.

Ex:- if statement, assignment statement, loop

Anonymous Function or Lambdas

A function without a name is called as Anonymous Function. It is also known as Lambda Function.

Anonymous Function are not defined using *def* keyword rather they are defined using *lambda* keyword.

Syntax:-

lambda argument_list : expression

Ex:-

lambda x : print(x)

lambda x, y : x + y

Creating a Lambda Function

Syntax:-

lambda argument_list : expression

Ex:-

: represents beginning of the function

Argument List

Expression

lambda x, y : x + y

lambda keyword represents an anonymous Function is being created

Calling Lambda Function

```
sum = lambda x : x + 1
sum(5)

 add = lambda x, y : x + y
add(5, 2)
```

Anonymous Function or Lambdas

- Lambda Function doesn’t have any Name

  Ex:- *lambda* x : print(x)

- Lambda function returns a function

  Ex:- show = *lambda* x : print(x)

- Lambda function can take zero or any number of argument but contains only one expression

  Ex:- *lambda* x, y : x + y

- In lambda Function there is no need to write return statement
- It can only contain expressions and can’t include statements in its body
- You can use all the type of Actual Arguments

Nested Lambda Function

When we write a lambda function inside another lambda function that is called nested lambda function.

```
add = lambda x=10 : (lambda y : x + y)
a = add()
print(a)
print(a(20))
```

Passing lambda Function to another Function

We can pass lambda function to another function.

```
def show(a):
        print(a(8))

show(lambda x: x)
```

Returning lambda Function

We can return a lambda function from function.

```
def add():
        y = 20
        return (lambda x : x+y)

a =add()
print(a(10))
```

Immediately Invoked Function Expressions (IIFE)

```
sum = lambda x : x + 1
sum(5)

(lambda x : x + 1)(5)

add = lambda x, y : x + y
add(5, 2)

(lambda x, y : x + y)(5, 2)
```

Function Decorator

A Decorator function is a function that accepts a function as parameter and returns a function.

A decorator takes the result of a function, modifies the result and returns it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

We use @function_name to specify a decorator to be applied on another function.