

# FUNCTIONS

## STUDY MATERIAL

- If a group of statements is repeatedly required then it is not recommended to write

these statements everytime seperately. We have to define these statements as a single unit and we can call that unit any number of times based on our requirement without rewriting. This unit is nothing but function.

- The main advantage of functions is code Reusability.
- Note: In other languages functions are known as methods, procedures, subroutines etc
  
- Python supports 2 types of functions
  - 1) Built in Functions
  - 2) User Defined Functions

## **1) Built in Functions:**

The functions which are coming along with Python software automatically, are called built in functions or pre defined functions.

Eg: id()

type()  
input()  
eval()  
etc..

## **2) User Defined Functions:**

The functions which are developed by programmer explicitly according to business requirements, are called user defined functions.

### **Syntax to Create User defined Functions:**

```
def function_name(parameters) :  
    """ doc string """  
    ----  
    ----  
    return value
```

**Note:** While creating functions we can use 2 keywords

- 1) def (mandatory)
- 2) return (optional)

**Eg 1:** Write a function to print Hello

**test.py**

| 1) def wish():

```
2) print("Hello Good Morning")
3) wish()
4) wish()
5) wish()
```

## Parameters

Parameters are inputs to the function. If a function contains parameters, then at the time of calling, compulsory we should provide values otherwise, otherwise we will get error.

**Eg:** Write a function to take name of the student as input and print wish message by name.

```
1) def wish(name):
2)     print("Hello",name," Good Morning")
3) wish("Durga")
4) wish("Ravi")
```

```
D:\Python_classes>py test.py
Hello Durga Good Morning
Hello Ravi Good Morning
```

**Eg:** Write a function to take number as input and print its square value

```
1) def squarelt(number):
2)     print("The Square of",number,"is", number*number)
3) squarelt(4)
4) squarelt(5)
```

```
D:\Python_classes>py test.py
The Square of 4 is 16
The Square of 5 is 25
```

## Return Statement:

Function can take input values as parameters and executes business logic, and returns output to the caller with return statement.

**Q) Write a Function to accept 2 Numbers as Input and return Sum**

```
1) def add(x,y):
2)     return x+y
3) result=add(10,20)
```

```
4) print("The sum is",result)
5) print("The sum is",add(100,200))
```

D:\Python\_classes>py test.py

The sum is 30

The sum is 300

If we are not writing return statement then default return value is None.

```
1) def f1():
2)     print("Hello")
3) f1()
4) print(f1())
```

### Output

Hello

Hello

None

### Q) Write a Function to check whether the given Number is Even OR Odd?

```
1) def even_odd(num):
2)     if num%2==0:
3)         print(num,"is Even Number")
4)     else:
5)         print(num,"is Odd Number")
6) even_odd(10)
7) even_odd(15)
```

### Output

D:\Python\_classes>py test.py

10 is Even Number

15 is Odd Number

### Q) Write a Function to find Factorial of given Number?

```
1) def fact(num):
2)     result=1
3)     while num>=1:
4)         result=result*num
5)         num=num-1
6)     return result
```

```
7) for i in range(1,5):
8)     print("The Factorial of",i,"is :",fact(i))
```

### Output

D:\Python\_classes>py test.py

The Factorial of 1 is : 1

The Factorial of 2 is : 2

The Factorial of 3 is : 6

The Factorial of 4 is : 24

## Returning Multiple Values from a Function:

In other languages like C,C++ and Java, function can return atmost one value. But in Python, a function can return any number of values.

### Eg 1:

```
1) def sum_sub(a,b):
2)     sum=a+b
3)     sub=a-b
4)     return sum,sub
5) x,y=sum_sub(100,50)
6) print("The Sum is :",x)
7) print("The Subtraction is :",y)
```

### Output

The Sum is : 150

The Subtraction is : 50

### Eg 2:

```
1) def calc(a,b):
2)     sum=a+b
3)     sub=a-b
4)     mul=a*b
5)     div=a/b
6)     return sum,sub,mul,div
7) t=calc(100,50)
8) print("The Results are")
9) for i in t:
10)    print(i)
```

### Output

The Results are

```
150
50
5000
2.0
```

## Types of Arguments

```
def f1(a,b):
    -----
    -----
    -----
f1(10,20)
```

a, b are formal arguments where as 10,20 are actual arguments.

There are 4 types are actual arguments are allowed in Python.

- 1) Positional Arguments
- 2) Keyword Arguments
- 3) Default Arguments
- 4) Variable Length Arguments

### 1) Positional Arguments:

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

```
def sub(a, b):
    print(a-b)
```

```
sub(100, 200)
sub(200, 100)
```

- The number of arguments and position of arguments must be matched. If we change the order then result may be changed.
- If we change the number of arguments then we will get error.

### 2) Keyword Arguments:

We can pass argument values by keyword i.e by parameter name.

- 1) `def wish(name,msg):`
- 2) `print("Hello",name,msg)`
- 3) `wish(name="Durga",msg="Good Morning")`
- 4) `wish(msg="Good Morning",name="Durga")`

### Output

Hello Durga Good Morning

Hello Durga Good Morning

Here the order of arguments is not important but number of arguments must be matched.

**Note:** We can use both positional and keyword arguments simultaneously. But first we have to take positional arguments and then keyword arguments, otherwise we will get syntaxerror.

- 1) `def wish(name,msg):`
- 2) `print("Hello",name,msg)`
- 3) `wish("Durga","GoodMorning")` → Valid
- 4) `wish("Durga",msg="GoodMorning")` → Valid
- 5) `wish(name="Durga","GoodMorning")` → Invalid
- 6) `SyntaxError: positional argument follows keyword argument`

### 3) Default Arguments:

Sometimes we can provide default values for our positional arguments.

- 1) `def wish(name="Guest"):`
- 2) `print("Hello",name,"Good Morning")`
- 3) `wish("Durga")`
- 4) `wish()`

#### Output

Hello Durga Good Morning

Hello Guest Good Morning

If we are not passing any name then only default value will be considered.

#### **\*\*\*Note:**

After default arguments we should not take non default arguments.

- 1) `def wish(name="Guest",msg="Good Morning"):` → Valid
- 2) `def wish(name,msg="Good Morning"):` → Valid
- 3) `def wish(name="Guest",msg):` → Invalid

`SyntaxError: non-default argument follows default argument`

### 4) Variable Length Arguments:

- Sometimes we can pass variable number of arguments to our function, such type of arguments are called variable length arguments.
- We can declare a variable length argument with \* symbol as follows
- `def f1(*n):`
- We can call this function by passing any number of arguments including zero number.
- Internally all these values represented in the form of tuple.

```
1) def sum(*n):
2)     total=0
3)     for n1 in n:
4)         total=total+n1
5)     print("The Sum=",total)
6)
7) sum()
8) sum(10)
9) sum(10,20)
10) sum(10,20,30,40)
```

### Output

The Sum= 0  
The Sum= 10  
The Sum= 30  
The Sum= 100

**Note:** We can mix variable length arguments with positional arguments.

```
1) def f1(n1,*s):
2)     print(n1)
3)     for s1 in s:
4)         print(s1)
5)
6) f1(10)
7) f1(10,20,30,40)
8) f1(10,"A",30,"B")
```

### Output

10  
10  
20  
30  
40  
10  
A  
30  
B

**Note:** After variable length argument, if we are taking any other arguments then we should provide values as keyword arguments.



```
1) def f1(*s,n1):
2)     for s1 in s:
3)         print(s1)
4)     print(n1)
5)
6) f1("A","B",n1=10)
```

### Output

A

B

10

f1("A","B",10) → Invalid

TypeError: f1() missing 1 required keyword-only argument: 'n1'

**Note:** We can declare key word variable length arguments also.

- For this we have to use \*\*.
- def f1(\*\*n):
- We can call this function by passing any number of keyword arguments. Internally these keyword arguments will be stored inside a dictionary.

```
1) def display(**kwargs):
2)     for k,v in kwargs.items():
3)         print(k,"=",v)
4) display(n1=10,n2=20,n3=30)
5) display(rno=100,name="Durga",marks=70,subject="Java")
```

### Output

n1 = 10

n2 = 20

n3 = 30

rno = 100

name = Durga

marks = 70

subject = Java

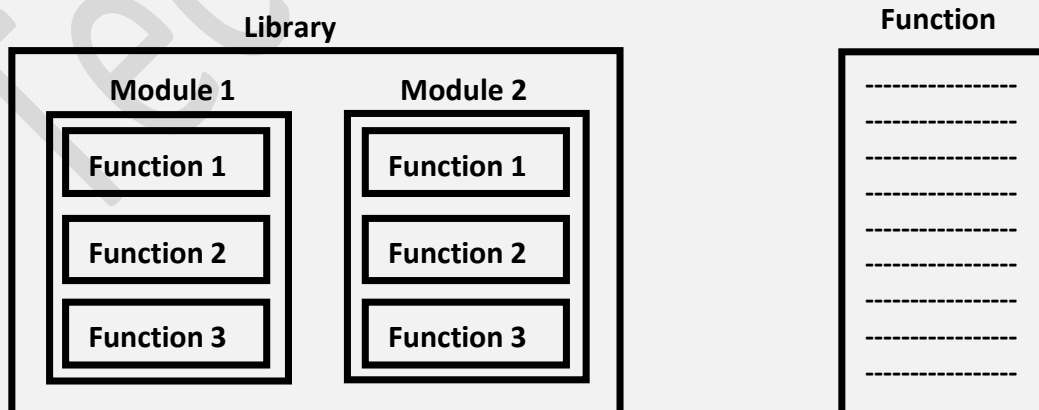
## Case Study:

```
def f(arg1,arg2,arg3=4,arg4=8):  
    print(arg1,arg2,arg3,arg4)
```

- 1)  $f(3,2) \rightarrow 3 \ 2 \ 4 \ 8$
- 2)  $f(10,20,30,40) \rightarrow 10 \ 20 \ 30 \ 40$
- 3)  $f(25,50,arg4=100) \rightarrow 25 \ 50 \ 4 \ 100$
- 4)  $f(arg4=2,arg1=3,arg2=4) \rightarrow 3 \ 4 \ 4 \ 2$
- 5)  $f() \rightarrow$  Invalid  
TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'
- 6)  $f(arg3=10, arg4=20, 30, 40) \rightarrow$  Invalid  
SyntaxError: positional argument follows keyword argument  
[After keyword arguments we should not take positional arguments]
- 7)  $f(4, 5, arg2 = 6) \rightarrow$  Invalid  
TypeError: f() got multiple values for argument 'arg2'
- 8)  $f(4, 5, arg3 = 5, arg5 = 6) \rightarrow$  Invalid  
TypeError: f() got an unexpected keyword argument 'arg5'

### Note: Function vs Module vs Library

- 1) A group of lines with some name is called a function
- 2) A group of functions saved to a file, is called Module
- 3) A group of Modules is nothing but Library



# Types of Variables

Python supports 2 types of variables.

- 1) Global Variables
- 2) Local Variables

## 1) Global Variables

- The variables which are declared outside of function are called global variables.
- These variables can be accessed in all functions of that module.

```
1) a=10 # global variable
2) def f1():
3)     print(a)
4)
5) def f2():
6)     print(a)
7)
8) f1()
9) f2()
```

### Output

```
10
10
```

## 2) Local Variables:

- The variables which are declared inside a function are called local variables.
- Local variables are available only for the function in which we declared it. i.e from outside of function we cannot access.

```
1) def f1():
2)     a=10
3)     print(a) # valid
4)
5) def f2():
6)     print(a) #invalid
7)
8) f1()
9) f2()
```

NameError: name 'a' is not defined

## global Keyword:

We can use global keyword for the following 2 purposes:

- 1) To declare global variable inside function
- 2) To make global variable available to the function so that we can perform required modifications

```
1) a=10
2) def f1():
3)     a=777
4)     print(a)
5)
6) def f2():
7)     print(a)
8)
9) f1()
10) f2()
11)
```

### Output

```
777
10
```

```
1) a=10
2) def f1():
3)     global a
4)     a=777
5)     print(a)
6) def f2():
7)     print(a)
8)
9) f1()
10) f2()
```

### Output

```
777
777
```

```
1) def f1():
2)     a=10
3)     print(a)
4)
5) def f2():
```

```
6) print(a)
7)
8) f1()
9) f2()
```

**Output:** NameError: name 'a' is not defined

```
1) def f1():
2)     global a
3)     a=10
4)     print(a)
5)
6) def f2():
7)     print(a)
8)
9) f1()
10) f2()
```

### Output

10  
10

**Note:** If global variable and local variable having the same name then we can access global variable inside a function as follows

```
1) a = 10 → Global Variable
2) def f1():
3)     a=777 → Local Variable
4)     print(a)
5)     print(globals()['a'])
6) f1()
```

### Output

777  
10

## Recursive Functions

A function that calls itself is known as Recursive Function.

**Eg:**

```
factorial(3) = 3 * factorial(2)
              = 3 * 2 * factorial(1)
              = 3 * 2 * 1 * factorial(0)
```

```
= 3 * 2 * 1 * 1
= 6
```

```
factorial(n) = n * factorial(n-1)
```

The main advantages of recursive functions are:

- 1) We can reduce length of the code and improves readability.
- 2) We can solve complex problems very easily.

## Q) Write a Python Function to find Factorial of given Number with Recursion

```
1) def factorial(n):
2)     if n==0:
3)         result=1
4)     else:
5)         result=n*factorial(n-1)
6)     return result
7) print("Factorial of 4 is :",factorial(4))
8) print("Factorial of 5 is :",factorial(5))
```

### Output

Factorial of 4 is : 24

Factorial of 5 is : 120

## Anonymous Functions:

- Sometimes we can declare a function without any name, such type of nameless functions are called anonymous functions or lambda functions.
- The main purpose of anonymous function is just for instant use (i.e. for one time usage)

## Normal Function:

We can define by using def keyword.

```
def squareIt(n):
    return n*n
```

## Lambda Function:

We can define by using lambda keyword `lambda n:n*n`

Syntax of lambda Function: `lambda argument_list : expression`

**Note:** By using Lambda Functions we can write very concise code so that readability of the program will be improved.

### Q) Write a Program to create a Lambda Function to find Square of given Number?

```
1) s=lambda n:n*n
2) print("The Square of 4 is :",s(4))
3) print("The Square of 5 is :",s(5))
```

#### Output

The Square of 4 is : 16

The Square of 5 is : 25

### Q) Lambda Function to find Sum of 2 given Numbers

```
1) s=lambda a,b:a+b
2) print("The Sum of 10,20 is:",s(10,20))
3) print("The Sum of 100,200 is:",s(100,200))
```

#### Output

The Sum of 10,20 is: 30

The Sum of 100,200 is: 300

### Q) Lambda Function to find biggest of given Values

```
1) s=lambda a,b:a if a>b else b
2) print("The Biggest of 10,20 is:",s(10,20))
3) print("The Biggest of 100,200 is:",s(100,200))
```

#### Output

The Biggest of 10,20 is: 20

The Biggest of 100,200 is: 200

**Note:** Lambda Function internally returns expression value and we are not required to write return statement explicitly.

**Note:** Sometimes we can pass function as argument to another function. In such cases lambda functions are best choice.

We can use lambda functions very commonly with filter(), map() and reduce() functions, because these functions expect function as argument.

## filter() Function:

We can use filter() function to filter values from the given sequence based on some condition.

`filter(function,sequence)`

Where Function Argument is responsible to perform conditional check Sequence can be List OR Tuple OR String.

### Q) Program to filter only Even Numbers from the List by using filter() Function?

#### Without Lambda Function:

```
1) def isEven(x):
2)     if x%2==0:
3)         return True
4)     else:
5)         return False
6) l=[0,5,10,15,20,25,30]
7) l1=list(filter(isEven,l))
8) print(l1) #[0,10,20,30]
```

#### With Lambda Function:

```
1) l=[0,5,10,15,20,25,30]
2) l1=list(filter(lambda x:x%2==0,l))
3) print(l1) #[0,10,20,30]
4) l2=list(filter(lambda x:x%2!=0,l))
5) print(l2) #[5,15,25]
```

## map() Function:

- For every element present in the given sequence, apply some functionality and generate new element with the required modification. For this requirement we should go for map() function.
- **Eg:** For every element present in the list perform double and generate new list of doubles.
- **Syntax:** map(function, sequence)



- The function can be applied on each element of sequence and generates new sequence.

### Without Lambda

```
1) l=[1,2,3,4,5]
2) def doublelt(x):
3)     return 2*x
4) l1=list(map(doublelt,l))
5) print(l1) #[2, 4, 6, 8, 10]
```

### With Lambda

```
1) l=[1,2,3,4,5]
2) l1=list(map(lambda x:2*x,l))
3) print(l1)  #[2, 4, 6, 8, 10]
```

-----  
**Eg 2:** To find square of given numbers

```
1) l=[1,2,3,4,5]
2) l1=list(map(lambda x:x*x,l))
3) print(l1)    #[1, 4, 9, 16, 25]
```

We can apply map() function on multiple lists also. But make sure all list should have same length.

**Syntax:** map(lambda x,y:x\*y,l1,l2))  
x is from l1 and y is from l2

```
1) l1=[1,2,3,4]
2) l2=[2,3,4,5]
3) l3=list(map(lambda x,y:x*y,l1,l2))
4) print(l3)    #[2, 6, 12, 20]
```

### reduce() Function:

- reduce() function reduces sequence of elements into a single element by applying the specified function.
- reduce(function,sequence)
- reduce() function present in functools module and hence we should write import statement.

```
1) from functools import *
2) l=[10,20,30,40,50]
```

```
3) result=reduce(lambda x,y:x+y,l)
4) print(result) # 150
```

**Eg:**

```
1) result=reduce(lambda x,y:x*y,l)
2) print(result) #12000000
```

**Eg:**

```
1) from functools import *
2) result=reduce(lambda x,y:x+y,range(1,101))
3) print(result) #5050
```

## Everything is an Object:

- In Python every thing is treated as object.
- Even functions also internally treated as objects only.

```
1) def f1():
2)     print("Hello")
3) print(f1)
4) print(id(f1))
```

**Output:**

```
<function f1 at 0x00419618>
4298264
```

## Function Aliasing:

For the existing function we can give another name, which is nothing but function aliasing.

```
1) def wish(name):
2)     print("Good Morning:",name)
3)
4) greeting=wish
5) print(id(wish))
6) print(id(greeting))
7)
8) greeting('Durga')
9) wish('Durga')
```

**Output:**

```
4429336
4429336
```

Good Morning: Durga  
Good Morning: Durga

### Note:

- In the above example only one function is available but we can call that function by using either wish name or greeting name.
- If we delete one name still we can access that function by using alias name.

```
1) def wish(name):
2)     print("Good Morning:",name)
3)
4) greeting=wish
5)
6) greeting('Durga')
7) wish('Durga')
8)
9) del wish
10) #wish('Durga') → NameError: name 'wish' is not defined
11) greeting('Pavan')
```

### Output:

Good Morning: Durga  
Good Morning: Durga  
Good Morning: Pavan

### Nested Functions:

We can declare a function inside another function, such type of functions are called Nested functions.

```
1) def outer():
2)     print("outer function started")
3)     def inner():
4)         print("inner function execution")
5)         print("outer function calling inner function")
6)         inner()
7)     outer()
8) #inner() → NameError: name 'inner' is not defined
```

### Output:

outer function started  
outer function calling inner function  
inner function execution

In the above example inner() function is local to outer() function and hence it is not possible to call directly from outside of outer() function.

**Note:** A function can return another function.

```
1) def outer():
2)     print("outer function started")
3)     def inner():
4)         print("inner function execution")
5)     print("outer function returning inner function")
6)     return inner
7) f1=outer()
8) f1()
9) f1()
10) f1()
```

**Output:**

```
outer function started
outer function returning inner function
inner function execution
inner function execution
inner function execution
```

**Q) What is the difference between the following lines?**

```
f1 = outer
f1 = outer()
```

- In the first case for the outer() function we are providing another name f1 (function aliasing).
- But in the second case we calling outer() function, which returns inner function. For that inner function() we are providing another name f1

**Note:** We can pass function as argument to another function

**Eg:** filter(function,sequence)  
map(function,sequence)  
reduce(function,sequence)