Functions:

- A function can contain a group of statements which performs the task
- Maintaining the code is an easy way
- Code reusability.
- There are two types of functions, Pre-defined or built-in functions,User-defined functions

Function related terminology:

- If we want to understand function concept in better way then we need to focus on function related terminology,
  - def keyword
  - name of the function
  - parenthesis ()
  - parameters (if required)
  - colon symbol:
  - function body
  - return type (optional)

Function parts:

- function definition or creation
- function calling

Based on parameters:

- Function without parameters
- Function with parameters.

Based on return statement:

- Function without return statement
- Function with return statement.

Defining a function:

- Very first step is we need to define a function, defining a function also called as creating a function.
- We need to use def keyword to create a function.
- After def keyword we should write name of the function.
- After function name, we should write parenthesis (),This parenthesis may contain parameters.
- If function having parameters, then we need to provide the values while calling.
- We will learn more in parameterized function
- After parenthesis we should write colon : symbol
- After : symbol in next line we should provide indentation
- Actual logic contains by function body
- This function body helps to perform the operation.

- Before closing the function, function may contain return type.
- **syntax:**

```
def functionname(parameters) :
    """ doc string"""
    Body of the function to perform operation
```

A naming convention to define a function:

- As discussed in Naming convention chapter, function name should be in lower case.
- If name having multiple words, then separating words with underscore (_) symbol is a good practice.

```
demo.py
# function creation

def display():
    print("welcome to function")
```

- When we execute above program, then function body not executed.
- To execute function body we need to call the function.

Calling a function:

- After function is created then we need to call that function to execute the function body.
- While calling the function, function name should be match otherwise we will get error.

```
demo.py
# function creation

def display():
    print("welcome to function concept")

# function calling

display()

output

welcome to function concept
```

Function without parameters:

- If a function having no parameters then that function is called as, a function without parameters
- A function without parameters also called, no parameterised function
- **syntax:**

```
def nameofthefunction():
    body of the function to perform operations

function calling
```

```
def display():
     print("Welcome to function which having no parameters")

# calling function
display()
output
Welcome to function which having no parameters
```

Function with parameters:

- If a function having parameters then that function called as function with parameters
- A function with parameters also called, parameterised function
- : If a function is parameterised then we need to pass values while calling that function otherwise we will get error
- There is no length limit for function parameters, means based on requirement we can provide any number of parameters

```
def testing(a):
     print("one parameterised function:", a)

testing(10)
output
one parameterised function: 10
```

A Function can call other function:

- Based on requirement a function can call another function as well.
- We can call a function inside another function.

```
demo.py

def m1():
     print("first function")

def m2():
     print("second function")
     m1()

m2()

output
second function
first function
```

return keyword in python:

- return is a keyword in python programming language.
- sed on return statement, functions can be divided into two types, Function without return, Function with return.
- By using return, we can return the result.
- If any function is not return anything, then by default that function returns None data type.
- We can also say as, if we are not writing return statement, then default return value is None

- Function can, Take input values, Process the logic, returns output to the caller with return statement
- **syntax:**

```
def nameofthefunction(parameter1, parameter2, …):
   body of the function
   return result
```

```
demo.py

   def add(a, b):
        c = a + b
        return c

   x=add(1, 2)
   print("Sum of two numbers is: ",x)

   output
   Sum of two numbers is:  3
```

A function can return multiple values:

- In python, a function can return multiple values
- If a function is returning multiple values then we can write return statement with multiple values
- If function is returning two values then while function calling we need to assign to two variable.
- If function is returning three values then while function calling we need to assign to three variables.
- **sysntax:**

```
def name_of_the_function():
    body of the function
    return value1, value2, value3,...,valueN

    demo.py

     def m1():
          a = 10
          b = 11
          return a,b

     #calling function

     x, y = m1()

     print("first value is:", x)
     print("second value is:", y)

     output

     first value is: 10
     second value is: 11
```

Functions are first class objects:

- Functions are considered as first-class objects

- When we create a function, the python interpreter internally creates an object
- In python, below things are possible to,
  - Assign function to variables

```
def add():
      print("We assigned function to variable ")

#Assign function to variable
sum=add

#calling function
sum()

output
We assigned function to variable
```

  - Pass function as a parameter to another function

```
def message():
     print("This is message function")
def display(x):
     print("This is display function")

message()
display(10)

output
This is message function
This is display function
```

  - Define one function inside another function

```
def first():
     print("This is outer function")
     def second():
          print("this is inner function")
     second()

#calling outer function

first()

output
This is outer function
this is inner function
```

  - Function can return another function

```
demo.py

def first():
     def second():
                print("A function is returning another
function”)
     return second

x=first()
```

```
x()

Output
A function is returning another function
```

Formal and actual arguments:

- When a function is defined it may have some parameters.
- These parameters receive the values
- Parameters are called as a 'formal arguments'
- When we call the function, we should pass values or data to the function
- These values are called as 'actual arguments'

```
demo.py

def add(a, b):
    c = a + b
    print(c)

# call the function
x = 10
y = 15
add(x, y)

output
25
```

- a and b called as formal arguments
- x and y called actual arguments

Types of arguments:

- In python there are 4 types of actual arguments are existing,
  - positional arguments
  - keyword arguments
  - default arguments
  - variable length arguments

Positional arguments:

- These are the arguments passed to a function in correct positional order
- The number of arguments and position of arguments should be matched, otherwise we will get error.

```
demo.py

def sub(x, y):
    print(x-y)

# calling function
sub(20, 10)

output
10
```

- If we change the number of arguments, then we will get error

- This function accepts only two arguments then if we are trying to provide three values then we will get error

```
demo.py

def sub(x, y):
     print(x-y)

# calling function
sub(10, 20, 30)

output
TypeError: sub() takes 2 positional arguments but 3 were
given
```

Keyword arguments:

- Keyword arguments are arguments that recognize the parameters by the name of the parameters.
- Example: Name of the function is cart(product, prirce)and parameters are product and price can be written as:
- At the time of calling this function, we must pass two values and we can write which value is for what by using name of the parameter,
- cart(product='bangles', price=20000) product and price are called as keywords in this scenario.Here we can change the order of arguments

```
demo.py

  def cart(product, price):
       print("product is :" , product)
       print("cost is :" , price)

  cart(product= "bangles ", price=20000)

  output
  Product is: bangles
  cost is: 200000
```

```
demo.py

def cart(product, price):
     print("product is :" , product)
     print("cost is :" , price)

cart(price=1200, product = "shirt")

output
Product is: shirt
```

- We can use both positional and keyword arguments simultaneously.
- But first we must take positional arguments and then keyword arguments, otherwise we will get syntax error.

```
demo.py

def details(id, name):
```

```
        print("Emp id is: ",id)
        print("Emp name is: ",name)

    details(1, name= "Sachin")

    output
    Emp id is: 1
    Emp name is: Sachin



    demo.py

     def details(id, name):
          print("Emp id is: ",id)
          print("Emp name is: ",name)

     details(name= "Sachin", 1)

     Error
        SyntaxError: positional argument follows keyword
argument
```

Default arguments:

- We can provide some default values for the function parameters in the definition.
- Let's take the function name as cart(product, price=40.0) (Example: In few shops any item is like a 40 rupees)
- Still if we provide the value at time of calling then default values will be override with passing value.
- If we are not passing any value, then only default value will be considered.

```
demo.py
def cart(product, price = 40.0):
     print("product is :" , product)
     print("cost is :" , price)

# calling function
cart(product = "pen ")

output

Product is: pen
Cost is : 40.0


 demo.py

 def cart(product, price = 40.0):
      print("product is :" , product)
      print("cost is :" , price)

 cart(product = "handbag ", price=10000)

 output
 Product is: handbag
```

```
Cost is : 10000


demo.py

 def cart(product, price = 40.0):
      print("product is :" , product)
      print("cost is :" , price)

 cart(price=500, product = "shirt")

 output

 Product is: shirt
 Cost is : 40.0



 demo.py

 def m1(a=10, b):
      print(a)
      print(b)

 # calling function
 m1(b=20)

 Error
      SyntaxError: non-default argument follows default
argument
```

Variable length arguments:

- If we define one parameterised function then during function calling we need to pass one value.
- If we define two parameterised function then during function calling we need to pass two values, if we pass more or less than two values then we will get error
- If we create variable length argument function then during function calling we can pass any number of values
  - totalcost(item1_cost, item2_cost) => totalcost(1, 2)# valid
  - totalcost(item1_cost, item2_cost) => totalcost(1, 2, 3)# Invalid
- To accept 'n' number of arguments, we need to use variable length argument.
- The variable length argument is an argument that can accept any number of values
- The variable length argument is written with a '*' (one star) before variable in function definition.
- **syntax:**

```
def nameofthefunction(x, *y):
     body of the function
```

- x is formal argument
- *y is variable length argument
- Now we can pass any number of values to this *y.

- Internally the provided values will be represented in tuple.

```
demo.py
  def m(*x):
       print(x)

m()
output
()
```

```
demo.py
def m(*x):
     print(x)

m(10, 20, 30)

output
(10, 20, 30)
```

Keyword variable length argument (**variable):

- **syntax:**

```
def m1(**kwargs):
     body of the function
```

- **kwargs represents as keyword variable argument
- Internally it represents like a dictionary object
- dictionary stores the data in the form of key value pairs.

```
demo.py

def print_kwargs(**kwargs):
     print(kwargs)

                    print_kwargs(id=1,  name="Sachin",
qualification="MCA")

Output
     {'id': 1, 'name': 'Sachin', 'qualification':
'MCA'}
```

Anonymous functions or Lambdas:

- A function without a name is called as anonymous function.
- Generally, to define normal function we need to use def keyword.
- To define anonymous function, we need to use lambda keyword.
- Lambda Function internally returns expression value and we no need to write return statement explicitly.
- **where to use:**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, these functions expect function as argument.

```
demo.py

s = lambda a: a*a
print(s)

output
<function <lambda> at 0x000001D3890F3E18>
```

```
demo.py

 add=lambda x, y: x+y
 result = add(1, 2)
 print('The sum of value is: ', result)

 output
 3
```

map() function:

- The map() function can apply the condition on every element which is available in the sequence of elements.
- After applying map function can generate new group of values
- **syntax:** map(function, sequence)

```
demo.py

without_gst_cost = [100,200,300,400]
            with_gst_cost = map(lambda x: x+10, without_gst_cost)
x=list(with_gst_cost)
                print("Without GST items costs: ",without_gst_cost)
print("With GST items costs: ",x)

output

Without GST items costs: [100, 200, 300, 400]
With GST items costs: [110, 210, 310, 410]
```

filter() function:

- Based on some condition we can filter values from sequence of values
- Where function argument is responsible to perform conditional check,sequence can be list or tuple or string.
- **syntax:** filter(function, sequence)

```
demo.py

items_cost = [999, 888, 1100, 1200, 1300, 777]
gt_thousand = filter(lambda x : x>1000, items_cost)
x=list(gt_thousand)
print("Eligible for discount: ",x)

output
Eligible for discount: [1100, 1200, 1300]
```

reduce() function:

- reduce() function reduces sequence of elements into a single element by applying the specific condition or logic.
- reduce() function present in functools module.
- So, to work with reduce we need to import functools module.
- **syntax:** reduce(function, sequence)

demo.py

```
from functools import reduce

each_items_costs = [111, 222, 333, 444]
        total_cost = reduce(lambda x, y: x+y, each_items_costs)
print(total_cost)
```

output

```
1110
```

Function Aliasing:

- We can give another name for existing function, this is called as function aliasing.

demo.py

```
def a():
    print("function - a")

b=a
b()
```

output

```
function - a
```

Function generators (yield keyword):

- Generators are just like functions that return a sequence of values.
- A generator function is written like an ordinary function, but this function uses yield keyword.
- We need to use for loop to iterate values one by one from generator object

yield keyword:

- yield is a keyword in python.
- We should use yield keywords with function and method, otherwise we will get error.

demo.py

```
print('Hello')
yield 100
```

```
output
SyntaxError: 'yield' outside function
```

```
dem.py

from sys import getsizeof

def m():
     return list(range(10))

x = m()
print("Values: ",x)
print("Size is: ",getsizeof(x))

output
Values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Size is: 200
```

```
dem.py

from sys import getsizeof

def m():
     yield list(range(1000))

x = m()
print(x)
print("Size is: ",getsizeof(x))

output
<generator object m at 0x000002765D985D58>
Size is: 88
```

Decorator: (Higher order function or Pure function):

- Decorator is a function.
- Decorator can accept another function as a parameter and returns the function
- Decorators are useful to perform some additional processing required by a function.
- decorator function takes parameter as another function
- We should define inner function inside the decorator function
- This function modifies or decorates the value of the function passed to the decorator function
- Return the inner function that has processed the value.
- After decorator created then we can call where are all required.

```
demo.py
  def m1(number):
       def m2():
           x=number()
           return x+2
      return m2
```

```
def m3():
     return 10


result = m1(m3)
r = result()
print(r)


output
12
```

@ symbol:

- Syntactic Sugar, Python allows you to simplify the calling of decorators using the @ symbol (this is called “pie” syntax).
- To apply decorator to any function we can use the @ symbol and decorator name on top of the function name.
- When we are using @ symbol then we no need to call decorator explicitly and pass the function name.

```
@decor
def m3():
    return 10
```

- We can call directly like, `print(m3())`

```
demo.py
    def m1(number):
        def m2():
             x=number()
             return x+2
         return m2

    @m1
    def m3():
         return 10

    x=m3()
    print(x)

    output
    12
```