FUNCTIONS

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
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

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")
5.
6.
7. D:\Python_classes>py test.py
8. Hello Durga  Good Morning
9. Hello Ravi  Good Morning
```

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

```
1. def squareIt(number):
2.     print("The Square of",number,"is", number*number)
3. squareIt(4)
4. squareIt(5)
5.
6. D:\Python_classes>py test.py
7. The Square of 4 is 16
8. 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))
6.
7.
8. D:\Python_classes>py test.py
9. The sum is 30
10. The sum is 300
```

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

Eg:

```
1. def  f1():
2.     print("Hello")
3. f1()
4. print(f1())
5.
6. Output
7. Hello
8. Hello
9. 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)
8.
9. Output
10. D:\Python_classes>py test.py
11. 10 is Even Number
12. 15 is Odd Number
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

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))
9)
10) Output
11) D:\Python_classes>py test.py
12) The Factorial of 1 is : 1
13) The Factorial of 2 is : 2
14) The Factorial of 3 is : 6
15) 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)
8)
9)  Output
10) The Sum is : 150
11) 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)
11)
12) Output
13) The Results are
14) 150
15) 50
16) 5000
17) 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.

Eg:

```
1. def wish(name,msg):
2.     print("Hello",name,msg)
3. wish(name="Durga",msg="Good Morning")
4. wish(msg="Good Morning",name="Durga")
5.
6. Output
7. Hello Durga Good Morning
8. 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.

```
def wish(name,msg):
        print("Hello",name,msg)
wish("Durga","GoodMorning")          ==>valid
wish("Durga",msg="GoodMorning")    ==>valid
wish(name="Durga","GoodMorning")  ==>invalid
SyntaxError: positional argument follows keyword argument
```

3. Default Arguments:

Sometimes we can provide default values for our positional arguments.

Eg:

```
1) def wish(name="Guest"):
2)     print("Hello",name,"Good Morning")
3)
4) wish("Durga")
5) wish()
6)
7) Output
8) Hello Durga Good Morning
9) 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

def wish(name="Guest",msg="Good Morning"): ===>Valid
def wish(name,msg="Good Morning"): ===>Valid
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.

Eg:

```
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)
11)
12) Output
13) The Sum= 0
14) The Sum= 10
15) The Sum= 30
16) The Sum= 100
```

Note:

We can mix variable length arguments with positional arguments.

Eg:

```
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")
9)
10) Output
11) 10
12) 10
13) 20
14) 30
15) 40
16) 10
17) A
18) 30
19) B
```

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

Eg:

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

Eg:

```
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")
6)
7)  Output
8)  n1 = 10
9)  n2 = 20
10) n3 = 30
11) rno = 100
12) name = Durga
13) marks = 70
14) subject = Java
```

Case Study:

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

1. f(3,2) ==> 3 2 4 8

2. f(10,20,30,40) ===>10 20 30 40

3. f(25,50,arg4=100) ==>25 50 4 100

4. f(arg4=2,arg1=3,arg2=4)===>3 4 4 2
5. f()===>Invalid
   TypeError: f() missing 2 required positional arguments: 'arg1' and 'arg2'

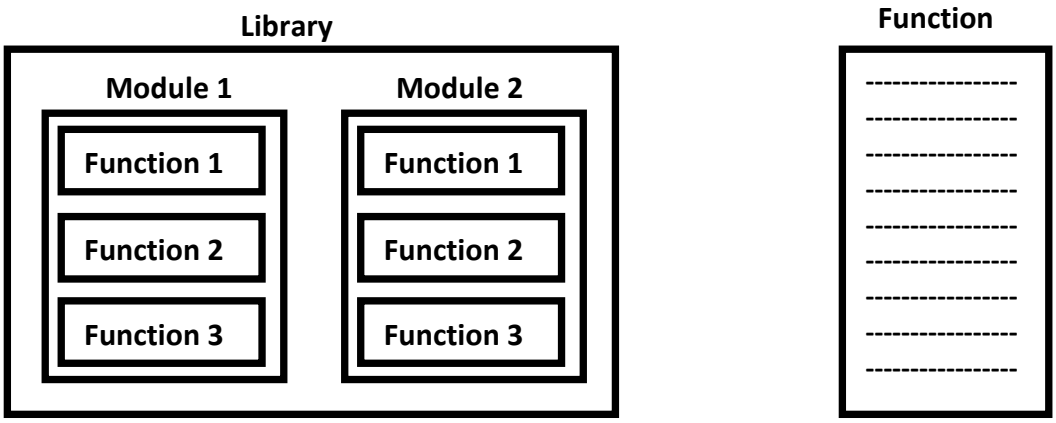
6. f(arg3=10,arg4=20,30,40) ==>Invalid
   SyntaxError: positional argument follows keyword argument
   [After keyword arguments we should not take positional arguments]

7. f(4,5,arg2=6)==>Invalid
   TypeError: f() got multiple values for argument 'arg2'

8. f(4,5,arg3=5,arg5=6)==>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.

Eg:

```
1)  a=10  # global variable
2)  def f1():
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
8)  f1()
9)  f2()
10)
11) Output
12) 10
13) 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.

Eg:

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

Eg 1:

```
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)
12) Output
13) 777
14) 10
```

Eg 2:

```
1)  a=10
2)  def f1():
3)      global a
4)      a=777
5)      print(a)
6)
7)  def f2():
8)      print(a)
9)
10) f1()
11) f2()
12)
13) Output
14) 777
15) 777
```

Eg 3:

```
1)  def f1():
2)      a=10
3)      print(a)
4)
5)  def f2():
6)      print(a)
7)
8)  f1()
9)  f2()
10)
11) NameError: name 'a' is not defined
```

Eg 4:

```
1)  def f1():
2)      global a
3)      a=10
4)      print(a)
5)
6)  def f2():
7)      print(a)
8)
9)  f1()
10) f2()
11)
12) Output
13) 10
14) 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()
7)
8)
9)  Output
10) 777
11) 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.

Eg:

```
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))
9)
10) Output
```

```
11) Factorial of 4 is : 24
12) 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))
4)
5)  Output
6)  The Square of 4 is : 16
7)  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))
4)
5)  Output
6)  The Sum of 10,20 is: 30
7)  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))
4)
5)  Output
6)  The Biggest of 10,20 is: 20
7)  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,b'z 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.

Eg: Without lambda

```
1)   l=[1,2,3,4,5]
2)  def doubleIt(x):
3)     return 2*x
4)  l1=list(map(doubleIt,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

Eg:

```
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.

Eg:

```
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
```

Note:

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

Eg:

```
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.

Eg:

```
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

Eg:

```
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.

Eg:

```
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.

Eg:

```
1)  def outer():
2)      print("outer function started")
3)      def inner():
4)          print("inner function execution")
```

```
    print("outer function returning inner function")
    return inner
f1=outer()
f1()
f1()
f1()
```

Output

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

Q. What is the differenece between the following lines?

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

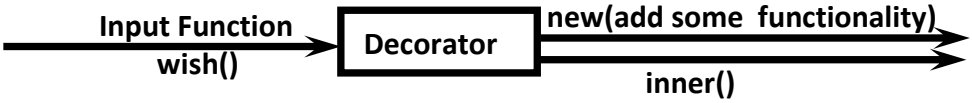
- In the first case for the outer() function we are providing another name f1(function aliasing).
- But in the second case we calling outer() function,which returns inner function.For that inner function() we are providing another name f1

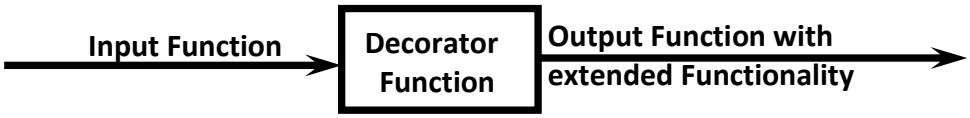
Note: We can pass function as argument to another function

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

Function Decorators:

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.





The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

```
1)  def wish(name):
2)    print("Hello",name,"Good Morning")
```

This function can always print same output for any name

Hello Durga Good Morning
Hello Ravi Good Morning
Hello Sunny Good Morning

But we want to modify this function to provide different message if name is Sunny.
We can do this without touching wish() function by using decorator.

Eg:

```
1)  def decor(func):
2)    def inner(name):
3)      if name=="Sunny":
4)        print("Hello Sunny Bad Morning")
5)      else:
6)        func(name)
7)    return inner
8)
9)  @decor
10) def wish(name):
11)   print("Hello",name,"Good Morning")
12)
13) wish("Durga")
14) wish("Ravi")
15) wish("Sunny")
16)
```

```
17) Output
18) Hello Durga Good Morning
19) Hello Ravi Good Morning
20) Hello Sunny Bad Morning
```

In the above program whenever we call wish() function automatically decor function will be executed.

How to call same function with decorator and without decorator:

We should not use @decor

```
1)  def decor(func):
2)      def inner(name):
3)          if name=="Sunny":
4)              print("Hello Sunny Bad Morning")
5)          else:
6)              func(name)
7)      return inner
8)
9)
10) def wish(name):
11)     print("Hello",name,"Good Morning")
12)
13) decorfunction=decor(wish)
14)
15) wish("Durga") #decorator wont be executed
16) wish("Sunny") #decorator wont be executed
17)
18) decorfunction("Durga")#decorator will be executed
19) decorfunction("Sunny")#decorator will be executed
20)
21) Output
22) Hello Durga Good Morning
23) Hello Sunny Good Morning
24) Hello Durga Good Morning
25) Hello Sunny Bad Morning
```

Eg 2:

```
1)  def  smart_division(func):
2)      def inner(a,b):
3)          print("We are dividing",a,"with",b)
4)          if b==0:
5)              print("OOPS...cannot divide")
6)              return
7)          else:
8)              return func(a,b)
```

```
9)      return inner
10)
11) @smart_division
12) def  division(a,b):
13)     return a/b
14)
15) print(division(20,2))
16) print(division(20,0))
17)
18) without decorator we will get Error.In this case output is:
19)
20) 10.0
21) Traceback (most recent call last):
22)   File "test.py", line 16, in <module>
23)     print(division(20,0))
24)   File "test.py", line 13, in division
25)     return a/b
26) ZeroDivisionError: division by zero
```

with decorator we won't get any error. In this case output is:

We are dividing 20 with 2
10.0
We are dividing 20 with 0
OOPS...cannot divide
None

Decorator Chaining

We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:
@decor1
@decor
def num():

For num() function we are applying 2 decorator functions. First inner decorator will work and then outer decorator.

Eg:

```
1)  def decor1(func):
2)      def inner():
3)          x=func()
4)          return x*x
```

```
5)      return inner
6)
7)  def decor(func):
8)      def inner():
9)          x=func()
10)         return 2*x
11)     return inner
12)
13) @decor1
14) @decor
15) def num():
16)     return 10
17)
18) print(num())
```

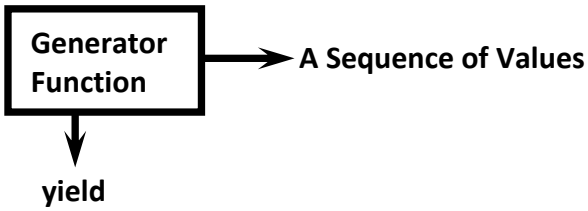
Demo Program for decorator Chaining:

```
1)  def decor(func):
2)      def inner(name):
3)          print("First Decor(decor) Function Execution")
4)          func(name)
5)      return inner
6)
7)  def decor1(func):
8)      def inner(name):
9)          print("Second Decor(decor1) Execution")
10)         func(name)
11)     return inner
12)
13) @decor1
14) @decor
15) def wish(name):
16)     print("Hello",name,"Good Morning")
17)
18) wish("Durga")
```

```
D:\durgaclasses>py decaratordemo1.py
Second Decor(decor1) Execution
First Decor(decor) Function Execution
Hello Durga Good Morning
```

Generators

Generator is a function which is responsible to generate a sequence of values.
We can write generator functions just like ordinary functions, but it uses yield keyword to return values.



Eg 1:

```
1)  def mygen():
2)      yield 'A'
3)      yield 'B'
4)      yield 'C'
5)
6)  g=mygen()
7)  print(type(g))
8)
9)  print(next(g))
10) print(next(g))
11) print(next(g))
12) print(next(g))
13)
14) Output
15) <class 'generator'>
16) A
17) B
18) C
19) Traceback (most recent call last):
20)   File "test.py", line 12, in <module>
21)     print(next(g))
22) StopIteration
```

Eg 2:

```
1)  def countdown(num):
2)      print("Start Countdown")
3)      while(num>0):
4)          yield num
5)          num=num-1
6)
7)  values=countdown(5)
8)  for x in values:
```

```
    print(x)

Output
Start Countdown
5
4
3
2
1
```

Eg 3: To generate first n numbers:

```
def firstn(num):
    n=1
    while n<=num:
        yield n
        n=n+1

values=firstn(5)
for x in values:
    print(x)

Output
1
2
3
4
5
```

We can convert generator into list as follows:

```
values=firstn(10)
l1=list(values)
print(l1)    #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eg 4: To generate Fibonacci Numbers...

The next is the sum of previous 2 numbers

Eg: 0,1,1,2,3,5,8,13,21,...

```
def fib():
    a,b=0,1
    while True:
        yield a
        a,b=b,a+b
for f in fib():
    if f>100:
        break
    print(f)
```

```
10)
11) Output
12) 0
13) 1
14) 1
15) 2
16) 3
17) 5
18) 8
19) 13
20) 21
21) 34
22) 55
23) 89
```

Advantages of Generator Functions:

1. when compared with class level iterators, generators are very easy to use
2. Improves memory utilization and performance.
3. Generators are best suitable for reading data from large number of large files
4. Generators work great for web scraping and crawling.

Generators vs Normal Collections wrt performance:

```
1)  import random
2)  import time
3)
4)  names = ['Sunny','Bunny','Chinny','Vinny']
5)  subjects = ['Python','Java','Blockchain']
6)
7)  def people_list(num_people):
8)      results = []
9)      for i in range(num_people):
10)         person = {
11)                 'id':i,
12)                 'name': random.choice(names),
13)                 'subject':random.choice(subjects)
14)                }
15)     results.append(person)
16)   return results
17)
18) def people_generator(num_people):
19)   for i in range(num_people):
20)      person = {
21)                 'id':i,
22)                 'name': random.choice(names),
23)                 'major':random.choice(subjects)
```

```
            }
        yield person

"""t1 = time.clock()
people = people_list(10000000)
t2 = time.clock()"""

t1 = time.clock()
people = people_generator(10000000)
t2 = time.clock()

print('Took {}'.format(t2-t1))
```

Note: In the above program observe the differnce wrt execution time by using list and generators

Generators vs Normal Collections wrt Memory Utilization:

Normal Collection:

```
l=[x*x for x in range(10000000000000000)]
print(l[0])
```

We will get MemoryError in this case because all these values are required to store in the memory.

Generators:

```
g=(x*x for x in range(10000000000000000))
print(next(g))
```

Output: 0
We won't get any MemoryError because the values won't be stored at the beginning

DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com