Functions in Python

What we will learn?

- Method vs Function
- Function Definition
- Function Call
- Function as First class object
- Call by Object Reference
- Formal and Actual Argumets

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

- Local and Global Variables
- Global keyword and globalsfunction
- Anonymous Functions
- Decorators
- Generators

Functions

> is a group of statements that are intended to perform a specific task.

> **Advantages**
>
> - Reusability
> - Modularity
> - Code Maintenance
> - Code Debugging
> - Reduces length of code

Difference between a Function and Method

> Function and method both look similar as they perform in an almost similar way, but the key difference is the concept of ‘Class and its Object’.

> Functions can be called only by its name, as it is defined independently.

> But methods cannot be called by its name only we need to invoke the class by reference of that class in which it is defined, that is, the method is defined within a class and hence they are dependent on that class.

> **objectname.methodname()**

> > > OR

> **classname().methodname()**

Defining a Function

- Keyword **def** that marks the start of the function header.
- A **function name** to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
- ** Parameters** (arguments) through which we pass values to a function. They are optional.
- ** A colon (:)** to mark the end of the function header.
- Optional documentation string (**docstring**) to describe what the function does.
- One or more valid **python statements** that make up the function body. Statements must have the same indentation level (usually 4 spaces).

> **Syntax**

> > **def** *function_name*(parameters)**:**

> > > """docstring"""

> > > statement(s)

```
def sum(a,b):
  '''Function to add two numbers'''
  print("Sum of {} and {} is {}".format(a,b,a+b))
```

Calling a Function

> Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
sum(10,15)
sum(10.3,21.4)
```

```
Sum of 10 and 15 is 25
Sum of 10.3 and 21.4 is 31.7
```

> The parameters a and b do not know which type of values they are going toreceive till the values are passed at the time of calling the function.

> This is **Dynamic Typing**, where type of data is determined only at runtime, not at compile time.

Returning Result from a Function

> The return statement is used to exit a function and go back to the place from where it was called.

> **Syntax of return:**
>
> > return [expression_list]

> This statement can contain an expression that gets evaluated and the value is returned.

> If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None object.** Examples:
>
> - return c
> - return 100
> - return lst
> - return a,b,c

```
def sum(a,b):
  return (a+b)

print(sum(100,200))
```

```
300
```

```
#Example 1 - Program to calculate factorial values of numbers
def factorial(n):
  """To calculate factorial of a number"""
  fact = 1
```

```
  while n>=1:
    fact *= n
    n -= 1

  return fact #outside while

#display factorial for first 8 numbers

for i in range(1,9):
  print("Factorial of {} is {}".format(i,factorial(i)))
```

```
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
Factorial of 7 is 5040
Factorial of 8 is 40320
```

```
#Example 2 - Python program to generates prime numbers with the help of a function to test pr
def prime(n):
  """Checks whether given number is prime or not"""
  x = 1
  for i in range (2,n):
    if n%i == 0:
      x=0
      break
    else:
      x=1
  return x


num = int(input("Enter number of prime numbers to be generated: "))
i=2
c=1
while True:
  if prime(i):
    print(i)
    c+=1
  i+=1
  if c>num:
    break
```

```
Enter number of prime numbers to be generated: 20
2
3
5
7
11
```

```
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
```

Returning multiple values from a function

```
# Example 3 - function returns addition and subtraction

def sum_sub(a,b):
  return a+b, a-b

#function call

x,y = sum_sub(10,5)
print(x, y)

z = sum_sub(10,5)
print(z)
```

```
15 5
(15, 5)
```

Function are **First class objects**

> It means functions are perfect objects in python

> Whenever we create a function, python interpreter internally creates an object.

> Since functions are objects, we can do following things with functions
>
> - Assign function to a variable
> - Define one function inside another function
> - To pass function as a parameter to another function
> - A function can return another function

Assign function to a variable

```
#Example 4
def display(str):
  return 'Python '+ str

x = display('Programming')
print(x)
```

```
Python Programming
```

Define one function inside another function

```
#Example 5
def func1(str):
  def func2():
    return 'Programming'
  return str + func2()
print(func1('Python '))
```

```
Python Programming
```

To pass function as a parameter to another function

```
#Example 6
def func1(str):
  return str + ' Programming'
def func2():
  return 'Python'

print(func1(func2()))
```

```
Python Programming
```

A function can return another function

```
#Example 7
def func1():
  def func2():
    return "Python Programming"
  return func2

x=func1()
print(x())
```

```
Python Programming
```

Pass by **Object Reference**

> In C and Java, Passing values to a function can be done in two ways,
>
> - **Pass by Value or Call by Value**
>
> - In this, copy of variable value is passed to the function and any modifications to that value will not reflect outside the function.
>
> - **Pass by Reference or Call by Reference**
>
> - Here, reference i.e memory address of the variable is passed.
> - The variable value is modified by the function through memory address and hence modified value will reflect outside the function also.

> Python uses a mechanism, which is known as "**Call-by-Object**", sometimes also called "**Call by Object Reference**" or "**Call by Sharing**".

> > If you pass immutable arguments like integers, strings or tuples to a function, the passing acts like call-by-value. The object reference is passed to the function parameters. They can't be changed within the function, because they can't be changed at all, i.e. they are immutable.

> > It's different, if we pass mutable arguments. They are also passed by object reference, but they can be changed in place in the function.

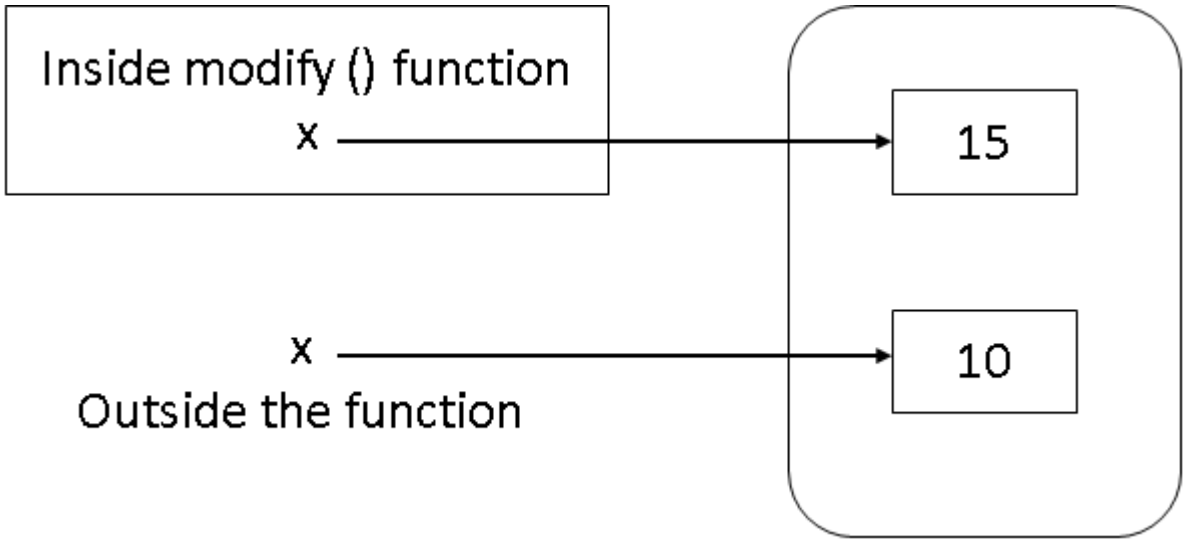
> > If we pass a list to a function, we have to consider two cases:
> >
> > - Elements of a list can be changed in place, i.e. the list will be changed even in the caller's scope.
> > - If a new list is assigned to the name, the old list will not be affected, i.e. the list in the caller's scope will remain untouched.

```
#Example 8 - Program to pass integer and modify it
def modify(x):
  x = 15
  print("Inside Function ",x,id(x))

x=10
```

```
modify(x)
print("Outside Function",x,id(x))
```

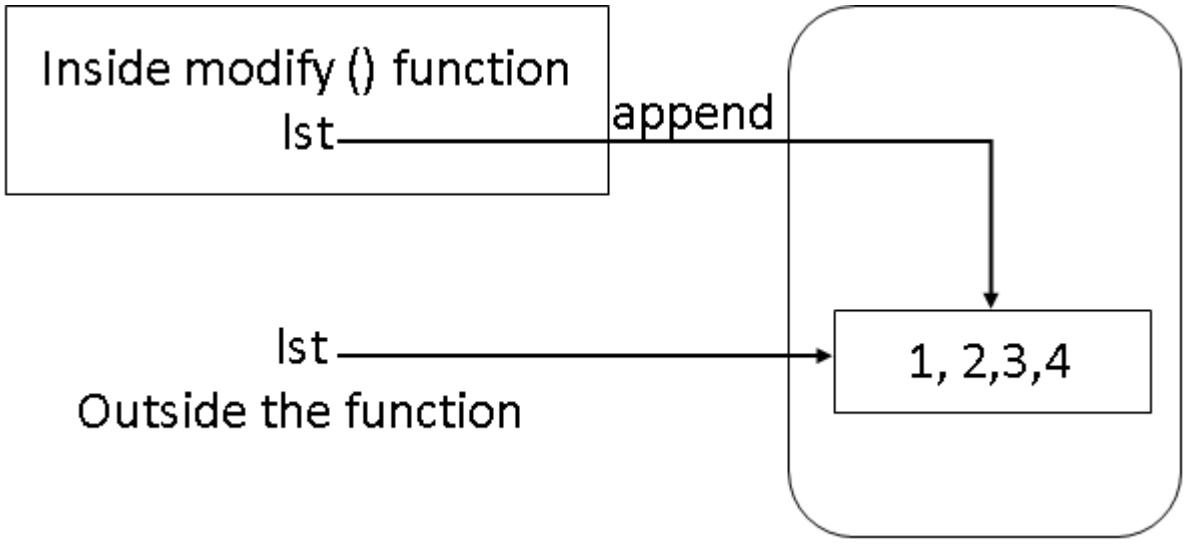
```
     Inside Function  15 94141076323264
     Outside Function 10 94141076323104
```




```
#Example 9 - Program to pass list and modify it
def modify(lst):
  lst.append(4)
  print("Inside Function ",lst,id(lst))

lst=[1,2,3]
modify(lst)
print("Outside Function",lst,id(lst))
```

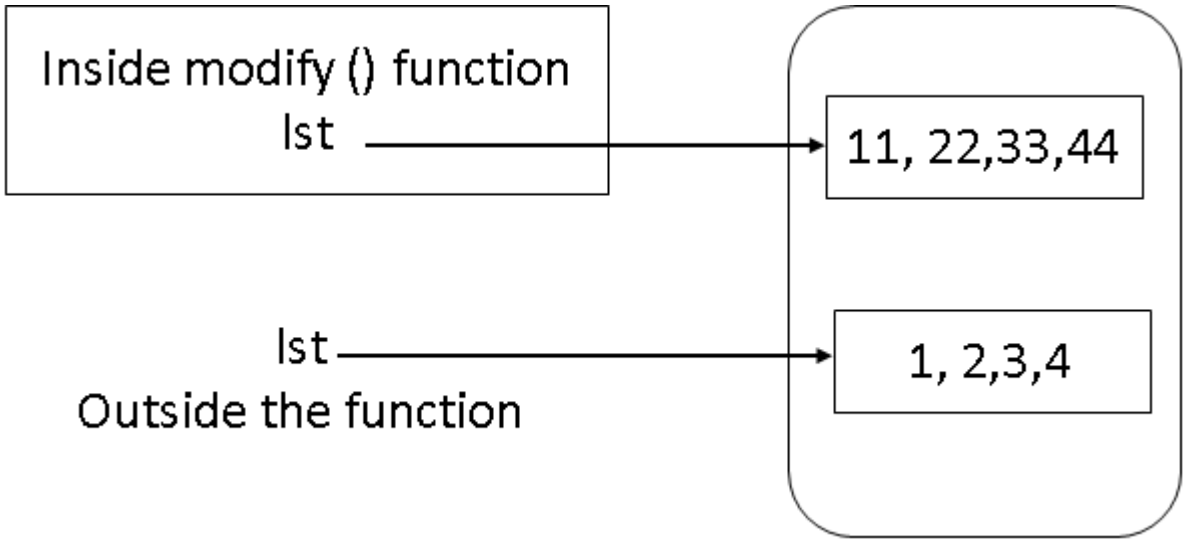
```
     Inside Function  [1, 2, 3, 4] 140342209095792
     Outside Function [1, 2, 3, 4] 140342209095792
```



```
#Example 10 - Program to create new object inside the function does not modify outside projec
def modify(lst):
  lst = [11,22,33,44]
  print("Inside Function ",lst,id(lst))

lst=[1,2,3,4]
modify(lst)
print("Outside Function",lst,id(lst))
```

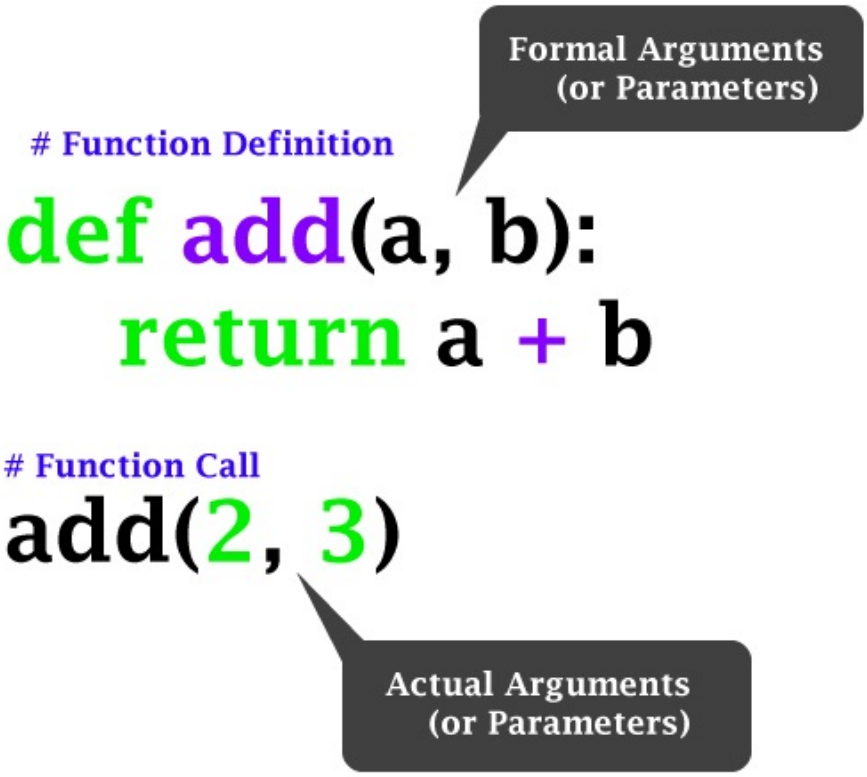
```
Inside Function  [11, 22, 33, 44] 140342209096112
Outside Function [1, 2, 3, 4] 140342209500576
```



Formal and Actual Arguments

> When a function is defined, there are some parameters which are useful for receiving values from outside the function. They are called **'Formal Arguments'.**

> When we make a function call, some values or data is passed to the function. These vaues are called actual arguments.



Four types of Actual arguments:

1. Positional Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable Length Arguments

Positional Arguments

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

The number of arguments and their position in the function definition should match exactly with the number and position of the arguments in the function call.

```
#Example 11 - Program to understand positional argument

def attach(s1,s2):
  '''Concatenate two string s1 and s2 and dislay concatenated string'''
  s3 = s1 + s2
  print(s3)

#function call
```

```
attach('Core','Python')
#there will be error raised, if we pass more than or less than 2 strings
#attach('Core','Python','Programming')
#attach('Core')
```

```
CorePython
```

Keyword Arguments

> These arguments are arguments that identifies the parameters by their names.

> Python allows functions to be called using keyword arguments. When we call functions in this way, **the order (position) of the arguments can be changed.**

```
# Examle 12 - Program to understand Keyword Arguments
def grocery(product, price):
  '''Display values of given arguments'''
  print('Product =',product)
  print('Price =',price)

#function call
grocery(product='Milk',price='70')
grocery(price='55.55', product='Oil')
grocery('Sugar', price='45.27')
#A keyword argument can follow a postional argument but vice cersa is not true.
#This statement will raise Syntax Error
#grocery(product='Sugar', 45.27)
```

```
Product = Milk
Price = 70
Product = Oil
Price = 55.55
Product = Sugar
Price = 45.27
```

Default Arguments

> It is an argument that assumes a default value if a value is not provided in the function call for that argument.

> Function arguments can have default values in Python. A default value can be provided to an argument by using the assignment operator (=)

```
#Example 13 - Program to understand default arguments
def grocery(product, price = 30.00):
```

```
    '''Display values of given arguments'''
    print('Product =',product)
    print('Price =',price)

#function call
grocery(product='Milk',price='70')
grocery('Rice',80.00)
grocery(product='Oil')
grocery('Sugar')
```

```
Product = Milk
Price = 70
Product = Rice
Price = 80.0
Product = Oil
Price = 30.0
Product = Sugar
Price = 30.0
```

Variable Length Arguments

> Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take variable number of arguments.

> Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an **arbitrary number of arguments.**

> In the function definition, we use an asterisk (*) before the parameter name to denote it as variable length argument.

> Syntax:

>> def func(farg, *args):

> When, we make a function call with multiple arguments, these arguments get wrapped up into a **tuple** before being passed into the function.

```
#Example 14 - Program to add 'n' numbers using variable length arguments
def add(n1, *n2):
  '''Add all the given numbers'''
  sum = 0
  for i in n2:
    sum += i

  print('Result =',n1+sum)
```

```
#function call
add(1,2,3,4,5)
add(1,2,3)
add(1)
#
lst= [1,2,3]
#add(4,lst) #Raises an arror as unsupported operand type(s)
add(4,*lst)#We can use * to unpack the list so that all elements of it can be passed as diffe
```

```
Result = 15
Result = 6
Result = 1
Result = 10
```

> **Keyword Variable Length Arguments** is an argument that can accept any number of values provided in the format of keys and values.

> If we want to use Keyword Variable Length Arguments, we can declare double asterik (**) before the argument.

> Syntax:

>> def func(farg, **kwargs):

> **kwargs represents keyword variable argument, which is internally represented as **dictionary object**.

```
#Example 15 - Program to for keyword variable length argument
def display(farg, **kwargs):   # **kwargs can take  or more values
  '''to display passed values'''
  print('Formal argument =', farg)

  for x,y in kwargs.items():   #The items() method returns a view object that displays a list
    print('Key =',x,', Value =',y)


#function calls
display(1,rno=10) #one formal argument and two keyword argument
display(2,rno=20,name='ABC') #one formal argument and four keyword argument
```

```
Formal argument = 1
Key = rno , Value = 10
Formal argument = 2
Key = rno , Value = 20
Key = name , Value = ABC
```

Local and Global Variables

> **Local Variable** - A variable whose scope is limited only to the function where it is created i.e a local variable value is available only in that function and not outside of that function.

> **Global Variable** - A varibale declared above a function. Such variables are available for all functions written after it.

```
#Example 16 - To understand scope of variable
a = 10
def func1():
  a = 20
  print("Inside Function, a =",a)

func1()
print("Outside Function, a =",a)
```

```
     Inside Function, a = 20
     Outside Function, a = 10
```

```
#Example 17 - To understand scope of variable
a = 10
def func1():
  b = 20
  print("Inside Function accessing global variable, a =",a)

func1()
#print("Outside Function accessing local varaible, b =",b)
```

```
     Inside Function accessing global variable, a = 10
```

The Global Keyword

> Sometimes, global variable and local variable may have the same name. In such cases, function, by default, refers to the local variable and ignores the global variable.

> If the programmer want to use the global variable inside a function, the keyword **global** can be use before the variable.

> In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

> **Basic rules for global** keyword in Python are:
>
> - When we create a variable inside a function, it is local by default.
> - When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
> - We use global keyword to read and write a global variable inside a function.
> - Use of global keyword outside a function has no effect.

```
#Example 18 - To understand global keyword (access global variable inside a function)
a = 10
def func1():
  #global a = 1   # Result in invalid syntax
  global a
  a = 20
  print("Inside Function, a =",a)

print("Outside Function, before func call, a =",a)
func1()
print("Outside Function, after func call a =",a)
```

```
Outside Function, before func call, a = 10
Inside Function, a = 20
Outside Function, after func call a = 20
```

The globals() function

> The globals() method returns the dictionary of the current global symbol table.

> A Global symbol table stores all information related to the global scope of the program, and is accessed in Python using globals() method.

> The global scope contains all functions, variables that are not associated with any class or function.

```
#Example 19 - Program to work with local and global variable of same name inside function bod
#Copy of global variable is created to work with
a = 10
b=1 #Global Var
def func1():
  a = 20   #Local var
  x = globals()['a'] # get value of global var 'a'
  print("Inside Function, global var 'a' =",x)
  print("Inside Function, local var 'a' =",a)
```

```
func1()
print("Outside Function, a =",a)
```

```
     Inside Function, global var 'a' = 10
     Inside Function, local var 'a' = 20
     Outside Function, a = 10
```

Anonymous Functions or Lambdas

> In Python, an anonymous function is a function that is defined **without a name.**

> While normal functions are defined using the def keyword in Python, anonymous functions are defined using the **lambda keyword**.Hence, anonymous functions are also called lambda functions.

> Format of lambda funcion: **lambda argument_list: expression**

> It can be observed from the format lambda function do not have any name.

> Unlike normal function, lambda function returs a function i.e. reference to a function object.

> Lamba function contains only one expression and they return results implicitly. Hence, no need of return statement.

```
#Example 19 - To create lambda funtion to return sum of two numbers
f = lambda x,y : x+y
print("Sum = ",f(1,2.5))
```

```
     Sum =  3.5
```

```
#Example 20 - To find maximum of two numbers
max = lambda x,y : x if x>y else y
print("Bigger number is ",max(11,22))
```

```
     Bigger number is  22
```

> **Use of Lambda Function in python**
>
> > We use lambda functions when we require a nameless function for a short period of time.

> > In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like **filter(), map()** etc.

> **Example use with filter()**

> > The filter() function in Python takes in a function and a list as arguments.

> > The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

> > Syntax: **filter(function, sequence)**

```
#Example 21 - To filter out only the even items from a list
list1 = [1,2,3, 4, 5, 6, 8, 11, 3, 12]
list2 = list(filter(lambda x: (x%2 == 0) , list1))
print(list2)

#             OR

def even(x):
  if x%2==0:
    return True
  else: return False

print(list(filter(even,list1)))
```

```
[2, 4, 6, 8, 12]
[2, 4, 6, 8, 12]
```

> **Example use with map()**

> > The map() function in Python takes in a function and a list.

> > The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

> > The function performs a specified operation on all elements of the sequence and the modified elements are returned which can be stored in another sequence.

> > Syntax: **map(function, sequence)**

```
#Example 22 - To double each item in a list using map()
```

```
#Example 22 - To double each item in a list using map()
list3 = [1, 5, 4, 6, 8, 11, 3, 12]

list4 = list(map(lambda x: x * 2 , list3))

print(list4)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

> **Example use with reduce()**
>
> > The reduce() function in Python reduces a sequence of elements to a single value by processing the elements to a function supplied.
> >
> > Syntax: **reduce(function, sequence)**

```
#Example 23 - To calculate product of element of a list
from functools import *
list5 = [1, 2, 3, 4]
res = reduce(lambda x,y: x * y , list5)
print(res)
```

```
24
```

Function Decorators

> Python has an interesting feature called decorators to add functionality to an existing code.
>
> This is also called metaprogramming because a part of the program tries to modify another part of the program at compile time.
>
> Decorators are like gift wrappers.
>
> If you want to extend the behavior of a function but don't want to modify it permanently, you can wrap a decorator on it.
>
> A decorator 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.
>
> Thus, decorators are useful to perform some additional processing required by a function.
>
> **Creating Decorator**

> > Decorators are **nested functions**. The outer function takes the function to decorate as an argument and then the inner function calls it.

```
#Example 24 -  Decorator that increments the value of a number by 2
def decor(func): #name of the decorator funtion
  def inner(): # This function actually modifies or decorates the value.
    value = func() # access the value return by func
    return value+2  # increase value by 2
  return inner # return the inner function that has processed or decorated the value.
```

> **How to use decorator?**
>
> > Once a decorator is created, it can be used for any function to decorate or process its result.

```
def num():
  return 10
res = decor(num)
print(res())
```

```
12
```

> > Python provides a much easier way for us to apply decorators using @ **symbol** and decorator name just above the function definition.

```
@decor             # apply decorator to the following function
def num1():
  return 100

print(num1())
```

```
102
```

> **Applying Multiple Decorators to a Single Function**
>
> > Multiple decorators can be applied to a single function. However, the decorators will be applied in the order in which they are called.

```
#Example 25 - Program to create and apply two decorators.
def uppercase_decor(func):
```

```
        def inner():
            res = func()
            res_uppercase = res.upper()
            return res_uppercase
        return inner

    def split_decor(func):
        def inner():
            res = func()
            res_splitted = res.split()
            return res_splitted
        return inner

    def bookname():
      return 'Core Python Programming'

    @split_decor
    @uppercase_decor
    def message():
      return 'Hello, Everyone'

    print(split_decor(uppercase_decor(bookname))())
    print(message())
```

```
['CORE', 'PYTHON', 'PROGRAMMING']
['HELLO,', 'EVERYONE']
```

> **Accepting Arguments in Decorator Functions**
>
> > Sometimes, there is a need to define a decorator that accepts arguments.
> >
> > This can be achieved by passing the arguments to the inner function.
> >
> > The arguments will then be passed to the function that is being decorated at call time.

```
#Example 26 - Program to create and apply decorator on function acceting arguments.
def decor(func):
  def inner(arg1):
    value = func(arg1)
    return value+2
  return inner

def square(x):
  return x*x
```

```
print(decor(square)(4))
```

```
18
```

Generators

> Generators are functions that returns sequence of values.

> It is written like an ordinary function but it uses yield statement. This statement is useful to return the value.

> If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

> The difference is that while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

```
#Example 27 - Program to create a generator that generates sequence of number from x to y

def gen1(x,y):
  while x<=y:
    yield x
    x+=1

g = gen1(2,8)  #fill generator object with numbers from 2 to 8
#display all numbers in generator using loop
print(type(g))
for i in g:
  print(i)

g = gen1(2,8)

#storing numbers of generator in a list
l=list(g)
print(l)
```

```
<class 'generator'>
2
3
4
5
6
7
```

```
8
[2, 3, 4, 5, 6, 7, 8]
```

> *next() * - can be used to retrieve element by element from a generator

```
g = gen1(1,4)
print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

```
1
2
3
4
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-34-7bd05950c0d6> in <module>()
      4 print(next(g))
      5 print(next(g))
----> 6 print(next(g))

StopIteration:
```

SEARCH STACK OVERFLOW