PYTHON DECORATORS:
Decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.
OR
Decorator is a function that can add additional functionality to an existing function.!

OUTLINE:
INPUT FUNCTION ==> DECORATOR FUNCTION ==> OUTPUT FUNCTION with Extended Functionality

Example:
```
def NorFun():
    print("Feature-1")
NorFun()
```

Example:
```
def DecFun(func):
    def Addon():
        func()
        print("Feature-2")
        print("Feature-3")
    return Addon

def NorFun():
    print("Feature-1")
NorFun=DecFun(NorFun) #=> Best for Debug
NorFun()
```

Example:
```
def DecFun(func):
    def Addon():
        func()
        print("Feature-2")
        print("Feature-3")
    return Addon

@DecFun
def NorFun():
    print("Feature-1")
NorFun() #=> Best for Debug
```

Example:
```
def NormalFun():
    print("Feature-1")
NormalFun()
```

Example:
```
def AddFun(func):
    def Inner():
        func()
        print("Feature-2")
    return Inner

def NormalFun():
    print("Feature-1")
```

```
NormalFun=AddFun(NormalFun)#=> Best for Debug
NormalFun()

Example:
def AddFun(func):
    def Inner():
        func()
        print("Feature-2")
    return Inner

@AddFun#Decorator
def NormalFun():
    print("Feature-1")
NormalFun()#=> Best for Debug

Example:
def MyDecor(func):
    def MyInner():
        print("Added New Functionality")
    return MyInner

@MyDecor
def MyOrginal():
    print("My Original Function")

MyOrginal()

Example:
def MyDecor(func):
    def MyInner():
        print("Added New Functionality")
        func()
    return MyInner

@MyDecor
def MyOrginal():
    print("My Original Function")
MyOrginal()

Example:
def Success(func):
    def Study():
        print("Prepare Well")
        func()
        print("Congratulations..!!")
    return Study

@Success
def MyFun():
    print("Say Hey You are PASS")
MyFun()

Example:
def Div(x,y):
    return x/y
print(Div(2,1))
```

```
print(Div(2,2))
```

Example:

```
def Div(x,y):
    return x/y
print(Div(2,1))
print(Div(2,0))#ZeroDivisionError
```

Example: with Decorator

```
def DivUpdate(func):
    def Inner(a,b):
        if b==0:
            print("SorryUnableToCompute")
        else:
            return func(a,b)
    return Inner

@DivUpdate #Callable
def Div(a,b):
    return a/b
print(Div(2,1))##=> Best for Debug
Div(2,0)
print(Div(2,2))
```

DivUpdate is a decorator, decorator function added some new functionality to the original function We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated.

Using Multiple Decorators:

```
def MyDecor_Two(func):
    def MyInner_Two():
        func()
        print("New Functionality for Decorator Two")
    return MyInner_Two

def MyDecor_One(func):
    def MyInner_One():
        func()
        print("New Functionality for Decorator One")
    return MyInner_One

@MyDecor_Two
@MyDecor_One
def MyOrginal():
    print("My Original Function")
MyOrginal()
```

What is a Python namespace?
A namespace is a system that has a unique name for each and every object in Python. An object might be a variable or a method. Python itself maintains a namespace in the form of a Python dictionary. Its Name (which means name, a unique identifier) + Space(which talks something related to scope).

Namespace & Variable Scope:
Names or Identifiers, To identify an object.

Example:
Variable Name
Function Name
Class Name
Method Name...!!

Namepsace is a system which control all the names which we use in our program. It allows us to reuse names in our program.

```
>>> x=4
x is a varible, 4 is the value
>>> x
>>> x=5
x is a varible, 5 is the value
>>> x
>>> x=4
>>> y=5
>>> x
>>> y
```

If we are writing small coding no issue we can maintain that uniqueness of variables use. In real time industry multiple programmers are developing code, there is a chance variable duplication.In that case namespace playing major rule.

```
Example:FirstModule.py
def HelloFun():
 print("Welcome to Namespace Classes-1")

Example:SecondModule.py
def HelloFun():
 print("Welcome to Namespace Classes-2")

Example:Main.py
import FirstModule
import SecondModule
FirstModule.HelloFun()
SecondModule.HelloFun()
```

NOTE:
In the above script function names are same but we can access using module names, this is exactly the use of namespace.

```
>>> dir()
>>> x=100
>>> dir()
```

There are 4 namespaces in Python ie LEGB Rule:
1 Local Scope
2 Global Scope
3. Enclosed
4. Builtin

1.Local Scope: We can access variables only within the function.
Example:

```
def InnerFunction():
    x=100
```

```
    print("X value is: ",x)
InnerFunction()
print("X value is: ",x)#NameError: name 'x' is not defined, OutOfScope

Example:
def Scope_Variables():
    x=200#Local Scope
    print(x+x)#Local+Local
    return
Scope_Variables()
```

2.Global Scope: We can access variables within the function & Out of the function

```
Example:
Example:
y=200
def InnerFunction():
    x=100
    print("X value is: ",x)
    print("Y value is: ",y)
InnerFunction()
print("Y value is: ",y)

Example:
x=40#Global Scope
def Scope_Variables1():
    x=200#Local Scope
    print(x+x)#Local+Local
    return
Scope_Variables1()
def Scope_Variables2():
    z=600#Local SCope
    print(x+z)#Global+Local
    return
Scope_Variables2()
```

Modifying the global variable in local scope

```
Example:
y=200
def InnerFunction():
    x=100
    y=y+1
    print("Inside X value is: ",x)
    print("Inside Y value is: ",y)
InnerFunction()
print("OutSide Y value is: ",y)
```

NOTE:
UnboundLocalError: local variable 'y' referenced before assignment

```
Example:
To modify global variable inside the local scope we need to use
'global' keyword.
y=200
def InnerFunction():
    x=100
    global y
```

```
    y=y+1
    print("Inside X value is: ",x)
    print("Inside Y value is: ",y)
InnerFunction()
print("OutSide Y value is: ",y)
```

3 Enclosing Scope:
Contains Names defined inside any and all enclosed functions.

Example:

```
y=20 #Global Scope
def OuterFunction():#Enclosed Function
    #Enclosed Scope
    #We can't access z outside the OuterFunction
    z=10
    def InnerFunction():#Inner/Nested Function
        #Local Scope
        x=5
        print("X Value is: ",x)
        print("Inside Y value is: ",y)
    InnerFunction()
    print("Z Value is: ",z)
OuterFunction()
```

Example:

```
y=20
def OuterFunction():
    z=100
    def InnerFunction():
        x=5
        z=z+2
        print("X Value is: ",x)
        print("Inside Z value is: ",z)
    InnerFunction()
    print("Z Value is: ",z)
OuterFunction()
```

NOTE:
UnboundLocalError: local variable 'z' referenced before assignment

Example:
To modify the enclosed variable in the local scope we need to use nonlocal keyword

```
y=20 #Globa;
def OuterFunction():
    z=100#Enclosed
    def InnerFunction():
        x=5#Local
        nonlocal z
        z=z+2
        print("X Value is: ",x)
        print("Inside Z value is: ",z)
    InnerFunction()
    print("Z Value is: ",z)
OuterFunction()
```

4. Builtin Scope:Contains names built-in to the Python language. All

these are already defined.
Example:

```
print("Welcome to Builting")
print(*range(2,10))
```

Example:

```
#Global Scope
x=1
def OuterFunction():
    #Enclosed Scope
    x=2
    def InnerFunction():
        #Local Scope
        x=3
        print("X value is: ",x)
    InnerFunction()
OuterFunction()
```

NOTE:
1. The above example display local value 3. Here PYTHON follows LEGB rule.
2. Python First search in Local, then Enclosed, then Global last Builtin
3. If not found it returns name error.