nonlocal

"nonlocal" is a keyword
Without using nonlocal keyword inner function can access local variable of outer function but it cannot update or modify. In order to update or modify inner function uses nonlocal keyword.

Syntax: nonlocal variable-list

After this statement, given list of variables are non local variables (local variables of outer function)

Example:

```
def fun1(): # Outer Function
    x=100 # Local Variable
    def fun2(): # Inner Function
        y=200 # Local Variable
        print(x)
        print(y)
    def fun3(): # Inner Function
        x=400 # Local Variable
        print(x)
    def fun4(): # Inner Function
        nonlocal x
        x=500
    fun2()
    fun3()
    fun4()
    print(x)

fun1()
```

Output

```
100
200
400
500
```

LEGB Rule

The LEGB rule in Python is a fundamental principle that dictates the order in which Python resolves names (variables, functions, classes, etc.) during program execution. LEGB stands for:

L - Local:

Python first searches for the name within the current function or method's local scope. This includes variables defined directly within the function and its parameters.

E - Enclosing Function Locals (or Enclosing):

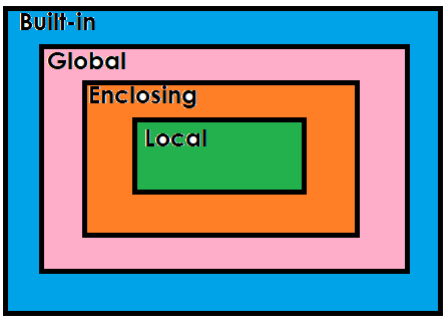
If the name is not found in the local scope, Python then looks in the local scopes of any enclosing functions, moving outwards from the innermost to the outermost enclosing function. This applies to nested functions.

G - Global:

If the name is still not found in the enclosing scopes, Python searches the global scope, which includes names defined at the module level (outside of any functions) or declared as global using the global keyword.

B - Built-in:

Finally, if the name is not found in any of the above scopes, Python checks the built-in scope. This scope contains names that are pre-defined in Python, such as print(), len(), range(), and various built-in exceptions.



Example:

```
def fun1(): # Outer Function
    x=100 # Local Variable
    def fun2(): # Inner Function
```

```
        y=200 # Local Variable
        print(x)
        print(y)
    def fun3(): # Inner Function
        x=400 # Local Variable
        print(x)
    def fun4(): # Inner Function
        nonlocal x
        x=500
    fun2()
    fun3()
    fun4()
    print(x)

fun1()
```

Output

Local Variable z=300
Non Local Variable y=200
Global Variable x=100

Decorator

Decorator is a special function in python.
Decorator is function which receives input as a function and return function as output.

Basic steps to work with decorator

1. Developing decorator
2. Applying decorator

decorator is applied using @decorator-name syntax

The advantage of decorators is,

1. Adding extra features to the existing function without modifying
2. Log information (Debugging)
3. Authentication and authorization

Decorators are 2 types

1. Function decorators
2. Class decorators

The decorators applied to function are called function decorators
The decorators applied to class are called class decorators

Basic steps for developing function decorators

1. Define a function with function as parameter
2. Define inner function which add or modify features of existing function or received function (wrapper function)
3. Return inner function/wrapper

Function decorators are 2 types.

1. Predefined function decorators
2. User defined function decorators

The decorators provided by python are called predefined function decorators.
Example: @staticmethod,@classmethod,@property,..

The decorators build by programmer are called user defined decorators, these are application specific.

Example:

```
def box(function):
    def wrapper():
        print("*"*30)
        function()
        print("*"*30)
    return wrapper


@box
def display():
    print("PYTHON PROGRAMMING")

@box
```

```
def print_info():
    print("PYTHON EASY LANGUAGE")

display()
print_info()
```

Output

```
******************************
PYTHON PROGRAMMING
******************************
******************************
PYTHON EASY LANGUAGE
******************************
```

Example:

```
import random
def otp_required(function):
    def wrapper():
        otp=random.randint(1000,9999)
        print(f'Login with {otp} Number')
        iotp=int(input("OTP No :"))
        if iotp==otp:
            function()
        else:
            print("Invalid Otp Number")
    return wrapper

@otp_required
def deposit():
    print("Inside Deposit")

@otp_required
def withdraw():
    print("Inside Withdraw")


deposit()
```

withdraw()

Output

```
Login with 1247 Number
OTP No :1245
Invalid Otp Number
Login with 9718 Number
OTP No :9718
Inside Withdraw
```

Example:

```
def smart_div(fun):
    def wrapper(a,b):
        if b==0:
            return float('inf')
        else:
            c=fun(a,b)
            return c
    return wrapper

@smart_div
def div(n1,n2):
    n3=n1/n2
    return n3


num1=int(input("Enter First Number :"))
num2=int(input("Enter Second Number :"))
num3=div(num1,num2)
print(f'{num1}/{num2}={num3}')
```

Output

```
Enter First Number :4
Enter Second Number :2
4/2=2.0

Enter First Number :5
```

Enter Second Number :0
5/0=inf

Closures

In Python, a closure is a nested function that remembers and has access to variables from its enclosing scope, even after the outer function has finished executing.

Key characteristics of a Python closure:

Nested Function: A closure involves an inner function defined within an outer function.
Access to Enclosing Scope: The inner function "closes over" variables from its outer (enclosing) function's scope. These are often referred to as "free variables" because they are not local to the inner function and are not passed as arguments.
Persistence of State: Even after the outer function has completed its execution and its local variables would typically be destroyed, the inner function (the closure) retains access to the values of those free variables. This allows the closure to maintain state across multiple calls.

Returned from Outer Function: The outer function typically returns the inner function object.

Example:

```
def find_power(num):
    def power(p):
        r=num**p
        return r
    return power



pow2=find_power(2)
res1=pow2(2)
```

```
res2=pow2(4)
res3=pow2(6)
print(res1,res2,res3)
pow5=find_power(5)
res4=pow5(2)
res5=pow5(4)
print(res4,res5)
res6=pow2(5)
print(res6)
```

Output

```
4 16 64
25 625
32
```

Example:

```
def draw_line(ch):
    def draw(size):
        print(ch*size)
    return draw


draw_stars=draw_line("*")
draw_dollar=draw_line("$")
draw_stars(20)
draw_dollar(10)
draw_stars(30)
draw_dollar(5)
```

Output

```
********************
$$$$$$$$$$
******************************
$$$$$
```

Closure is an inner function which performs operations used data of outer function.