Inner function can access local variables of outer function directly but it cannot modify or update local variables of outer function directly.
Local variables of outer function are called non local variables.

Example:
```
def fun1():
    x=100 # Local Variable
    def fun2():
        x=200 # Create Local Variable in fun2
        print(x)


    fun2()
    print(x)

fun1()
```

Output
```
200
100
```

nonlocal keyword

nonlocal keyword is used modify or update local variable of outer function inside inner function. Accessing local variable of outer function can be done directly inside inner function but it cannot update or modify without nonlocal keyword.

Syntax: nonlocal variable-name, variable-name, variable-name

The listed variables are identified as non local variables (local variables of outer function)
Listed variables should not be used before nonlocal statement.

Example:
```
def fun1():
    x=100
    def fun2():
        nonlocal x
        x=200
        print(x)

    fun2()
    print(x)

fun1()
```

Output

```
200
200
```

LEGB Rule

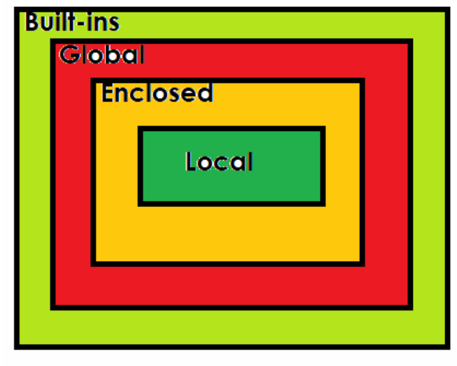
In Python, the LEGB rule dictates the order in which the interpreter searches for variable names. It is an acronym that stands for Local, Enclosing function locals, Global, and Built-in. When a variable is referenced, Python searches these scopes in order:

Local (L):
This scope includes names defined within the current function or block of code.

Enclosing function locals (E):
If the name is not found locally, Python searches the scopes of any enclosing functions, moving outwards.

Global (G):
This scope encompasses names defined at the top level of a module or declared using the global keyword.

Built-in (B):
Finally, if the name is still not found, Python checks the built-in namespace, which contains predefined functions and constants.



Example:

```
x=100 # Global Variable
def fun1():
    y=200 # Local Variable of fun1
    def fun2():
        z=300 # Local Variable of fun2
        print(z)
        print(y)
        print(x)
        print(__name__)

    fun2()

fun1()
```

Output

```
300
200
100
__main__
```

Example:

```
def calculator(n1,n2,opr):
    res=None
    def add():
        return n1+n2
    def sub():
        return n1-n2
    def multiply():
        return n1*n2
    def div():
        return n1/n2
    match(opr):
        case '+':
          res=add()
        case '-':
          res=sub()
        case '*':
          res=multiply()
        case '/':
          res=div()

    return res


#main
num1=int(input("Enter First Number :"))
num2=int(input("Enter Second Number :"))
opr=input("Enter Operator :")
result=calculator(num1,num2,opr)
print(result)
```

Output

```
Enter First Number :10
Enter Second Number :5
Enter Operator :/
2.0

Enter First Number :9
Enter Second Number :3
Enter Operator :*
27
```

Decorators

What is decorator in python?
Decorator is a special function in python.

Decorator is a function which receives input as function and return output as another function.
In Python, a decorator is a function that takes another function as an argument, extends its behavior without explicitly modifying it, and returns the modified function. Decorators provide a way to add functionality to functions or methods.

Applications of decorators

1. Logging
2. Access Control

Decorators are two types

1. Predefined decorators
2. User defined decorators

Predefined decorators
The existing decorators are called predefined decorators and these decorators are provided by python.
@staticmethod, @classmethod, @abstractmethod,..

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

Basic steps for working with decorators

1. Developing decorator
2. Applying decorator

Decorator is applied to function or method using @decorator-name

Example:

```
def draw_line(fun):
   def update_fun():
      print("*"*40)
      fun()
      print("*"*40)
   return update_fun

@draw_line
def display():
    print("PYTHON LANGUAGE")

@draw_line
def print_msg():
    print("DECORATOR TEST")

display()
print_msg()
```

Output

```
****************************************
PYTHON LANGUAGE
****************************************
****************************************
DECORATOR TEST
****************************************
```

Example:

```
def decorator(f):
    def update_fun():
        print("updated fun")
    return update_fun

@decorator
def fun1():
    print("fun1")

@decorator
def fun2():
    print("fun2")

fun1()
fun2()

#internally mech
#x=decorator(fun2)
#x()
```

Output

```
updated fun
```

updated fun

Example:
```
def authentication(function):
    def update_fun():
        user=input("UserName :")
        pwd=input("Password :")
        if user=="nit" and pwd=="nit":
            function()
        else:
            print("Invalid username or password")
    return update_fun

@authentication
def withdraw():
    print("withdraw")

@authentication
def deposit():
    print("deposit")


deposit()
withdraw()
```

Output
```
UserName :nit
Password :n123
Invalid username or password
UserName :nit
Password :nit123
Invalid username or password

UserName :nit
Password :nit
deposit
UserName :nit
Password :nit
Withdraw
```

Decorator chaining
Applying multiple decorators to function is called decorator chaining.

Example:
```
def decorator2(f):
    def update_fun():
        print("Decorator2")
        f()
    return update_fun

def decorator1(f):
    def update_fun():
        print("Decorator1")
        f()
    return update_fun

@decorator2
@decorator1
def fun1():
    print("Function1")


fun1()
```
Output
```
Decorator2
Decorator1
Function1
```