Nested Functions/Inner functions

A function inside function is called nested function.

Nested functions are used,

1. Dividing functionality of one function into number of sub function
2. Decorators
3. Closures

Nested function can perform operation using the data of outer function

Nested function can access local variables of outer function but outer function cannot access local variables of inner function.

Nested function is used within outer function but cannot accessible outside outer function.

Example:

```
def fun1():
    print("outer function")
    def fun2():
        print("inner function/nested function")
    fun2()

def main():
    fun1()

main()
```

Output:

```
outer function
inner function/nested function
>>>
```

Example:

```
def fun1():
    x=100 # local variable
    def fun2():
        print("x=",x) # "x" is variable of fun1/outer function

    fun2()

def main():
    fun1()
```

```
main()
```

Output:

```
x= 100
>>>
```

Example:

```
def fun1():
    def fun2():
        x=100 # local variable of fun2

    fun2()
    print(x)

def main():
    fun1()
main()
```

Output:

```
NameError: name 'x' is not defined
```

LEGB

Python search names/variables using one searching method called LEGB.
LEGB stands,
L → Local
E → Enclosed Block
G → Global
B → Builtins module

Example:

```
x=100 # global variable
def fun1():
    y=200 # local variable
    def fun2():
```

```
        z=300 # local variable
        print(x,y,z,sep="\n")
        print(__name__)

    fun2()

def main():
    fun1()

main()
```

Output:

```
100
200
300
__main__
```

nonlocal

The nonlocal statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. This is important because the default behavior for binding is to search the local namespace first.

```
nonlocal  variable-name,variable-name,…
```

Example:

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

def main():
    fun1()
```

```
main()
```

Output:

```
100
200
>>>
```

Example:

```
def calculator(n1,n2,opr):
    res=0
    def add():
        nonlocal res
        res=n1+n2
    def sub():
        nonlocal res
        res=n1-n2
    def multiply():
        nonlocal res
        res=n1*n2
    def div():
        nonlocal res
        res=n1/n2

    if opr=='+':
       add()
    if opr=='-':
        sub()
    if opr=='*':
        multiply()
    if opr=='/':
        div()
    return res

def main():
    num1=int(input("Enter first number"))
    num2=int(input("Enter second number"))
    opr=input("Enter operator")
    result=calculator(num1,num2,opr)
    print(result)
```

```
main()
```

Output:

```
Enter first number5
Enter second number2
Enter operator+
7
>>>
========= RESTART: C:/Users/user/Desktop/python6pm/py146.py
=========
Enter first number100
Enter second number50
Enter operator-
50
>>>
```

Decorators

A function returning another function, usually applied as a function transformation using the @wrapper syntax. Common examples for decorators are classmethod() and staticmethod().

Decorators are used to extend the functionality of existing function without modifying it.

Decorators are build using nested function

Decorator function receive a function as input

And transform into another function using nested function

And return nested function

Syntax:

```
def <function-name>(function):
        def <nested-function-name>([args]):
                statement-1
                statement-2
                may include functionality of old function
        return <nested function>
```

Example:

```
def box(f):
    def new_fun1():
        print("********************")
        f()
```

```
        print("**********")
    return new_fun1
@box
def fun1():
    print("Hello Python")

def main():
    #nf=box(fun1)
    #nf()
    fun1()

main()
```

Output:

```
**********
Hello Python
**********
>>>
```