Decorator chaining

Chaining decorators involves applying multiple decorators to a single function. Python allows you to chain decorators by stacking them on top of each other, and they are executed from the innermost to the outermost decorator.

In Python, a decorator is a special construct that allows us to add extra functionality to an existing function or class without modifying its source code. A decorator is a callable that takes another function or class as input and returns a modified version of it.

Example:

```
def draw_dollars(function):
    def display_dollars():
        print("$"*40)
        function()
        print("$"*40)
    return display_dollars


def draw_stars(function):
    def display_stars():
        print("*"*40)
        function()
        print("*"*40)
    return display_stars


@draw_dollars
@draw_stars
def display():
    print("PYTHON")

display()
```

Output

$$
****
PYTHON
****
$$

Closures

A Closure in Python is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure’s copies of their values or references, even when the function is invoked outside their scope.

Closure is inner function which uses data of outer function to perform operations even outer function execution completed.

Closures avoid using global variables.

```
def closure():
    a=10
    b=5
    def add():
        return a+b

    return add


c=closure()
result=c()
```

```
Output
15
```

```
print(result)
```

Example

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


p5=power(5)
res1=p5(2)
res2=p5(3)
res3=p5(5)
print(res1,res2,res3)
p7=power(7)
res4=p7(2)
res5=p7(5)
print(res4,res5)
res6=p5(4)
print(res6)
```

Output

```
25 125 3125
49 16807
625
```

Example

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


draw_stars=draw_line("*")
draw_dollar=draw_line("$")
draw_cap=draw_line("^")

draw_stars(10)
draw_stars(15)
draw_dollar(20)
draw_dollar(30)
draw_cap(5)
draw_cap(12)
```

Output

```
**********
***************
$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$
^^^^^
^^^^^^^^^^^^
```

Example

```
def calculator(num1,num2):
```

Output

```
7 3 10 2.5
```

```
    def calc(opr):
        if opr=='+':
            return num1+num2
        elif opr=='-':
            return num1-num2
        elif opr=='*':
            return num1*num2
        elif opr=='/':
            return num1/num2
    return calc


calculator1=calculator(5,2)
res1=calculator1('+')
res2=calculator1('-')
res3=calculator1('*')
res4=calculator1('/')
print(res1,res2,res3,res4)
```

Generator function or generators

Generators are used for creating iterator objects

Generator functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

Generators in Python are used to create iterators and return a traversal object. It helps in traversing all the items one at a time with the help of the keyword yield.

Normal function returns value using return keyword.

Generator function returns value using yield keyword.

What is difference between return and yield?

return	**yield**
This keyword return value, after returning value it terminates execution of function.	Yield keyword return value, after returning value, it pause execution of function. when iterated it resume back and continue where it is paused.
Function with return keyword is not generator function	Function with yield is called generator function, which return iterator object.