Python

Experiment No. 8

Aim: To implement concepts of function decorators

Problem Statements:

1. Program to implement a decorator and repeat the function 5 times through 1 call
2. Program to create even_arg_decorator that checks if one argument passed to a function is even, if it is even allow function to execute else print error message.
3. Program to create decorator to multiply the output by a variable amount.
4. Program to create two chain decorators square and doubled (multiply number by 2) and test the decorated function with calculate(num)

Theory:

In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.
The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

Prerequisites for learning decorators
Before we learn about decorators, we need to understand a few important concepts related to Python functions. Also, remember that everything in Python is an object, even functions are objects.

Nested Function
We can include one function inside another, known as a nested function. For example,

```
def outer(x):
  def inner(y):
    return x + y
  return inner

add_five = outer(5)
result = add_five(6)
print(result)  # prints 11

# Output: 11
```

Pass Function as Argument
We can pass a function as an argument to another function in Python. For Example,

```
def add(x, y):
   return x + y

def calculate(func, x, y):
   return func(x, y)

result = calculate(add, 4, 6)
print(result)  # prints 10
```

Return a Function as a Value

In Python, we can also return a function as a return value. For example,

```
def greeting(name):
   def hello():
      return "Hello, " + name + "!"
   return hello

greet = greeting("Atlantis")
print(greet())  # prints "Hello, Atlantis!"

# Output: Hello, Atlantis!
```

Python Decorators

A Python decorator is a function that takes in a function and returns it by adding some functionality.

In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

Code:

```
def make_pretty(func):
    def inner():
        print("I got decorated")
    func()
    return inner

def ordinary():
    print("I am ordinary")
# decorate the ordinary function
decorated_func = make_pretty(ordinary)
```

```
# call the decorated function
decorated_func()
```

Output:

```
I am ordinary
I got decorated
```

In the example shown above, make_pretty() is a decorator.

@ Symbol With Decorator

Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol.
For example,

Code:

```
def make_pretty(func):
    def inner():
        print("I got decorated")
    func()
    return inner

@make_pretty
def ordinary():
    print("I am ordinary")

ordinary()
```

Output:

```
I am ordinary
I got decorated
```

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:
Code:

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
```

```
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner

@smart_divide
def divide(a, b):
    print(a/b)

divide(2,5)

divide(2,0)
```

Output:

```
I am going to divide 2 and 5
0.4
I am going to divide 2 and 0
Whoops! cannot divide
```

Chaining Decorators in Python

Multiple decorators can be chained in Python.

To chain decorators in Python, we can apply multiple decorators to a single function by placing them one after the other, with the most inner decorator being applied first. The order in which we chain decorators matter.

Code:

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 15)
        func(*args, **kwargs)
        print("*" * 15)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 15)
        func(*args, **kwargs)
        print("%" * 15)
    return inner


@star
@percent
def printer(msg):
    print(msg)
```

```
printer("Hello")
```

Output:

```
**************
/o/o/o/o/o/o/o/o/o/o/o/o/o/
/o/o/o/o/o/o/o/o/o/o/o/o/o/
Hello
/o/o/o/o/o/o/o/o/o/o/o/o/o/
/o/o/o/o/o/o/o/o/o/o/o/o/o/
**************
```

Q.1 Program to implement a decorator and repeat the function 5 times through 1 call

Code:

```
def outer(func):
    def inner():
      print("Hello")
      func()
    return inner

@outer
def world():
    print("World")
    return world

for i in range(5):
    print("-----")
    world()
```

Output:

```
-----
Hello
World
-----
Hello
World
-----
Hello
World
-----
Hello
World
-----
Hello
World
```

Q.2 Program to create even_arg_decorator that checks if one argument passed to a function is even, if it is even allow function to execute else print error message.

Code:

```
n = int(input("Enter a number : "))
def check(func):
    def inner(x):
        if x%2 == 0:
            return func(x)
        else:
            print("Error!")
    return inner

@check
def inputNum(num):
    return "Even"

inputNum(n)
```

Output:

```
Enter a number : 1
Error!
```

Q.3 Program to create decorator to multiply the output by a variable amount.

Code:

```
num = int(input("Enter a number : "))
def multiply(num):
    def mul(func):
        def inner(x):
            return num*func(x)
        return inner
    return mul

@multiply(10)
def product(num):
    return num
product(num)
```

Output:

```
Enter a number : 123
1230
```

Q.4 Program to create two chain decorators square and doubled (multiply number by 2) and test the decorated function with calculate(num)

Code:

```
num = int(input("Enter a number : "))
def square(func):
    def inner(x):
        return func(x)**2
    return inner

def double(func):
    def inner(x):
        return func(x)*2
    return inner

@square
@double

def calculate(num):
    return num
calculate(num)
```

Output:

```
Enter a number : 6
144
```

Conclusion: Thus studied function decorators