Python fundamentals day 48

Today's Agenda

- Decorators contd…
- Closures

Decorators contd…

Now our expectation is to take an input list and first find the square of each element and then find the product of it. Let's see how to do it

```
def power_of(ref):

    def wrapper(lst):
        lst=list(map(lambda x:x**2, lst))
        ref(lst)
    return wrapper

def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

mod_get_product=power_of(get_product)
mod_get_product([1,2,3,4,5])
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
14400
```

We can do the same in a simpler way

```
def power_of(ref):

    def wrapper(lst):
        lst=list(map(lambda x:x**2, lst))
        ref(lst)
    return wrapper

@power_of
def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

get_product([1,2,3,4,5])
```

Output:

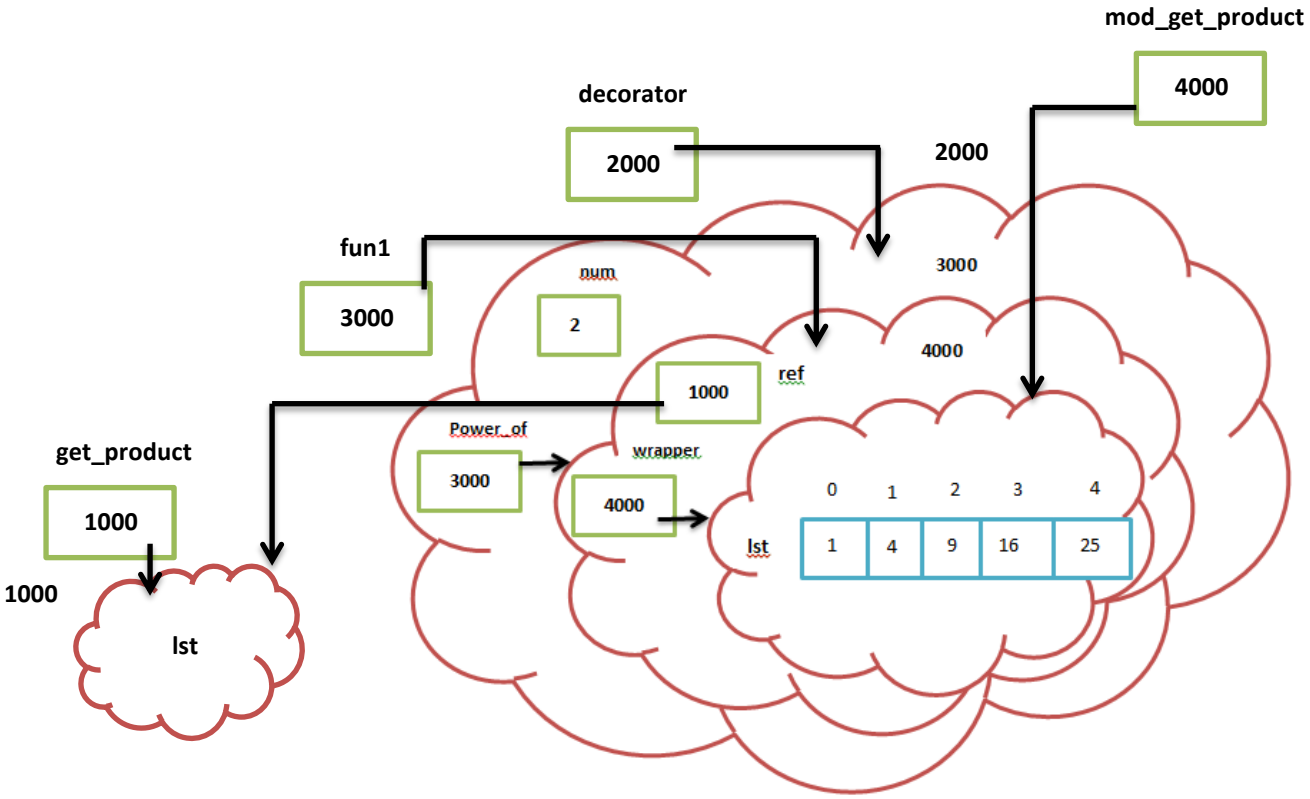
```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
14400
```

power_of will automatically consider the power to be 2. But what if we want it to be any other number? Let us see

```
def decorator(num):

    def power_of(ref):

        def wrapper(lst):
            lst=list(map(lambda x:x**num, lst))
            ref(lst)
        return wrapper
    return power_of

def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

fun1=decorator(3)
mod_get_product=fun1(get_product)
mod_get_product([1,2,3,4,5])
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
1728000
```



Above code can also be written as

```
def decorator(num):

    def power_of(ref):

        def wrapper(lst):
            lst=list(map(lambda x:x**num, lst))
            ref(lst)
        return wrapper
    return power_of

@decorator(3)
def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

get_product([1,2,3,4,5])
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
1728000
```

Closures

Though we have seen the inner/outer function there is another tricky thing to know. Let us see what it is

```
def outer():
    x=99

    def inner():
        print(x)
    return inner

fun=outer()
fun()
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
```

Now we are trying to delete the outer function. Let us see what happens

```
def outer():
    x=99

    def inner():
        print(x)
    return inner

fun=outer()
del outer
fun()
# will get the output because of concept called as closures
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
```

We get the output even after deleting the oyter function because of the concept called as closures.

When the outer function is deleted and the inner function is still present, though the variables in outer function got deleted the inner function will still have the access to those variables, because somewhere in memory the values are still present.

Let us look at another example to understand correctly

```
def outer():
    x=99

    def inner1():
        y=88

        def inner2():
            print(x)
            print(y)
        return inner2
    return inner1

fun1=outer()
fun2=fun1()
fun2()
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
88
```

Now let us try deleting the outer functions

```
def outer():
    x=99

    def inner1():
        y=88

        def inner2():
            print(x)
            print(y)
        return inner2
    return inner1

fun1=outer()
fun2=fun1()
fun2()
del outer
del fun1
fun2()
```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
88
99
88
```