Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #66
Decorators

By,
PCPS Team-2022
Prof. Apoorva MS

Verified by,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Introduction

Decorators are a powerful and useful tool in Python since it allows programmers to **modify the behavior of a function or a class without changing the source code of it**. It wraps another function in order **to extend the behavior of wrapped function**, **without permanently modifying wrapped function.** In Decorators, functions are taken as the argument into another function and then called inside the wrapper function. Using decorators, we can extend the features of different functions in a common way.

Decorators can be used in following scenarios:

- **Logging:** Decorators can be used to log the execution of a function. This can be useful for **debugging and tracking performance.**
- **Performance testing:** Decorators can be used to measure the execution time of a function. This can be useful for **optimizing the performance.**
- **Caching:** Decorators can be used to cache the results of a function. This can be useful for **improving performance and reducing database load.**
- **Memoization:** Decorators can be used to memoize the results of a function. This can be useful **for improving performance by avoiding unnecessary calculations**.
- **Authorization:** Decorators can be used to control access to a function. This can be useful for **implementing security features.**

Few fictitious examples to understand the **implementation of decorators** are here.

Example_code_1: Simple two functions

```
def deco_f1(fn):
   def inner_function():
      print("before f1")
      fn();       print("after f1")
   return inner_function
def f1():
   print("in f1")
df1 = deco_f1()
df1()
#f1()
```

```
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/d.py",
line 8, in <module>
    df1 = deco_f1()
TypeError: deco_f1() missing 1 required positional argument: 'fn'
```

In the above code, if there is a call to f1() directly, think about what gets printed? See the output below.

Output:
```
in f1
```

Idea of decorator is to call f1() and it must execute the decorator function first and then execute f1() which is the decorated function. If this is the requirement, then **@ must be used to say that deco_f1 is the decorator function that must be executed when f1() is called.**

Example_code_2:
```
def deco_f1(fn):
   def inner_function():
      print("before f1")
      fn()
      print("after f1")
   return inner_function
@deco_f1    # added only this now. No other change
def f1():
   print("in f1")
f1()
```

Output:
```
before f1
in f1
after f1
```

Note: @ indicates that we are applying decorator function to decorated function. It can also be written without using @ symbol as shown below.

Example_code_3:
```
def deco_f1(fn):
   def inner_function():
      print("before f1")
      fn()
      print("after f1")
   return inner_function
def f1():
   print("in f1")
f = deco_f1(f1)
f()
```

Output:
```
before f1
in f1
after f1
```

Think about this -> Does the code look like a closure or a callback?

Now let us try to understand where the decorators are used with few requirements from the client.

Requirement #1: Consider an example to find the product of all elements in the given list.

```
Li = [1, 2, 3, 4, 5]
def calculate_product(Li):
        product = 1
        for item in Li:
            product *= item
        return product
result = calculate_product(Li)
print(result)
```

Output:
120

Requirement #2:

Now the client wants to extend this with an **additional functionality, i.e., to find the total time taken to find the product**. So we can write the new code which includes this requirement or we can use decorators. We shall look at both the ways.

Version 1: Without using decorator

```
import time
def calculate_product(Li):
        product = 1
        time.sleep(2)
        for item in Li:
            product *= item
        return product
start_time = time.time()
Li = [1, 2, 3, 4, 5]
result = calculate_product(Li)
end_time = time.time()
time_taken = end_time - start_time
print("Time taken by function",time_taken)
print("Result:", result)
```

Output:

```
Time taken by function 2.0001144409179688
Result: 120
```

Now think about, which function must be the decorator function? Which must be the decorated function?

Version 2: Using the decorator

Additional functionality can be treated like a wrapper function/**Decorator** function and **calculate_product can be treated as the function being decorated**.

```
import time
def measure_execution_time(func):
        def wrapper(l):
            start_time = time.time();          result = func(l)
            end_time = time.time() ;           execution_time = end_time - start_time
            print("The time taken by function ",execution_time)
            return result
        return wrapper
Li = [1, 2, 3, 4, 5]
@ measure_execution_time
def calculate_product(Li):
        product = 1
        for item in Li:
            product *= item
        return product
result = calculate_product(Li)
print(result)
```

Output:

```
The time taken by function  0.0
120
```

Can we apply the **same decorator function on multiple functions**? **Yes.**

Example_code_4: The function calculate() is a decorator function which is decorating three functions namely factorial, sqrt, maximum

```
import math
def calculate(f): #decorator function
        def inner1(*args): #*args is variable argument
            print("decorator")
            f(*args) # this is being decorated by decorator
              print("**************")
         return inner1
@calculate
def factorial(num):
        print("In factorial function",math.factorial(num))
@calculate
def sqrt(num):
        print("In sqrt function",math.sqrt(num))
@calculate
def maximum(*num):
print("In maximum function",max(num[0],num[1],num[2]))
factorial(5) ;sqrt(16) ;maximum(23,9,78)
```

Output:

```
decorator
In factorial function 120
**************
decorator
In sqrt function 4.0
**************
decorator
In maximum function 78
**************
```

Chaining Decorators - Decorating a function with multiple decorators

It is also possible that multiple decorators can be applied on single function i.e., we can add multiple features to the existing function by using different decorators.

Example_code_5:

```
def deco_x(fn):
   def my_function():
      print("X"*20)
      fn()
      print("X"*20)
   return my_function

def deco_y(fn):
   def my_function():
      print("Y"*20)
      fn()
      print("Y"*20)
   return my_function

@deco_x
@deco_y
def say_hello():
   print("hello")
deco_x(deco_y(say_hello())) #same as usage of @deco above
```

Output:

```
XXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYY
hello
YYYYYYYYYYYYYYYYYYYY
XXXXXXXXXXXXXXXXXXXX
```

Advantages of using Decorators

- **Code reusability:** Decorators promote code reusability by encapsulating behavior that can be applied to multiple functions
- **Extensibility:** Decorators can be easily extended or modified without changing the original function or method's source code.
- **Debugging and logging made easy:** Decorators can be used to make debugging and logging easier. For example, you can use a decorator to log the input and output of a function, and then use that decorator to help you debug the function.

Example_code_6:

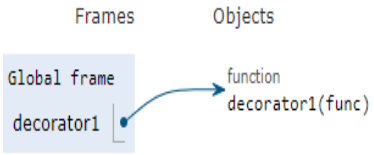
```
def decorator1(func):#decorator function
    def inner1():
        print("this is before decoration ")
        func() # this is being decorated by decorator
        print("this is after decoration ")
    return inner1

@decorator1
def function (): #this is the function which is being decorated
    print("This is the original function")
    #function = decorator1(function)
# calling the function
function()#calls decorated function
```

Output:

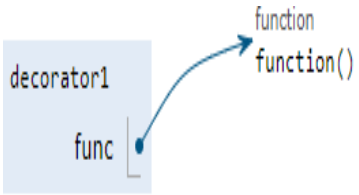
```
this is before decoration
This is the original function
this is after decoration
```

Visualization using python tutor:

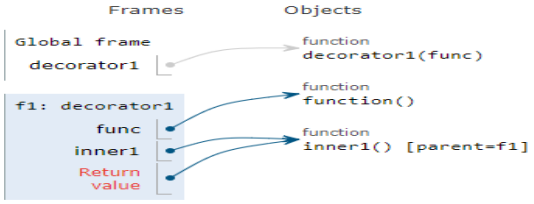
1. decorator() is processed



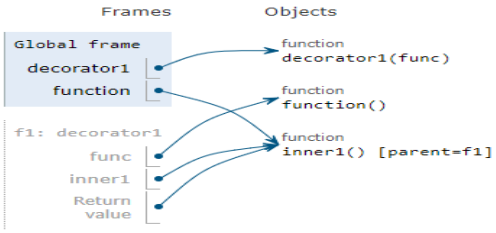
2.decorator() is called and executed



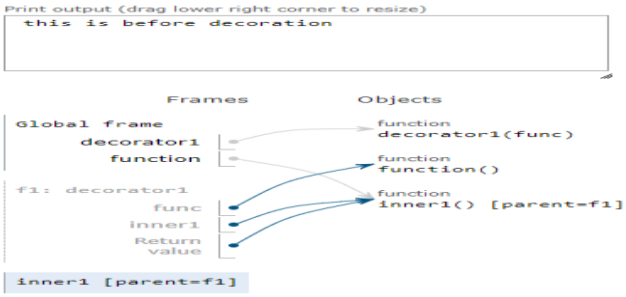
3.inner() is defined and reference to it is returned



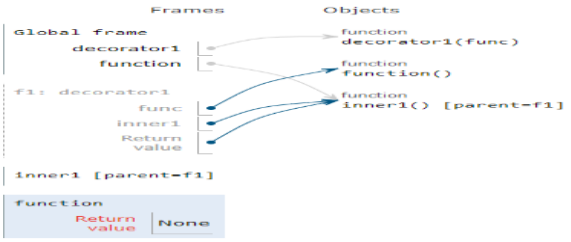
4. function() is called which in turn calls inner1()



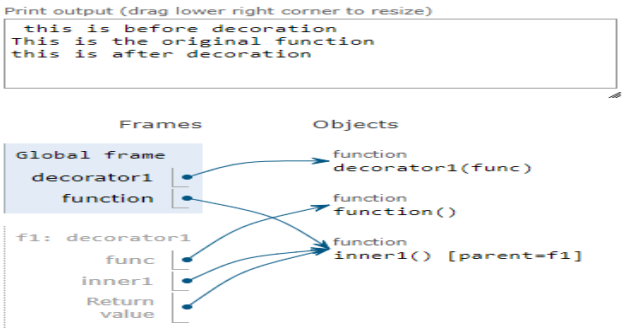
5.inner1() is executed and displays the string " this is before decoration"



6. Then inner1() is calls func() and displays the string "This is the orginal function"



7. Final Step



-END–