Decorators in Python

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behaviour of function or class.

Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. But before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

First Class Objects

In Python, functions are **first class objects** that means that functions in Python can be used or passed as arguments.

Properties of first class functions:

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Consider the below examples for better understanding.

Example 1: Treating the functions as objects.

```
def shout(text):
    return text.upper()

print(shout('Hello'))

yell = shout

print(yell('Hello'))
```

Output:

```
HELLO
HELLO
```

In the above example, we have assigned the function shout to a variable. This will not call the function instead it takes the function object referenced by shout and creates a second name pointing to it, yell.

Example 2: Passing the function as argument

```
# Python program to illustrate functions can be passed as arguments to other functions
```

```
def shout(text):

    return text.upper()

def whisper(text):

    return text.lower()

def greet(func):

    # storing the function in a variable

greeting = func("""Hi, I am created by a function passed
as an argument.""")

print (greeting)



greet(shout)

greet(whisper)
```

Output:

```
HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.
hi, i am created by a function passed as an argument.
```

In the above example, the greet function takes another function as a parameter (shout and whisper in this case). The function passed as argument is then called inside the function greets.

Example 3: Returning functions from another functions.

```
# Python program to illustrate functions

# Functions can return another function
```

```
def create_adder(x):
    def adder(y):
        return x+y
    return adder

add_15 = create_adder(15)
print(add_15(10))
```

Output:

25

In the above example, we have created a function inside of another function and then have returned the function created inside.

As stated above the decorators are used to modify the behaviour of function or class. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax for Decorator:

```
@gfg_decorator
def hello_decorator():
    print("Gfg")

'''Above code is equivalent to -

def hello_decorator():
    print("Gfg")
```

```
hello_decorator = gfg_decorator(hello_decorator)'''
```

In the above code, `gfg_decorator` is a callable function, will add some code on the top of some another callable function, `hello_decorator` function and return the wrapper function.

Decorator can modify the behavior:

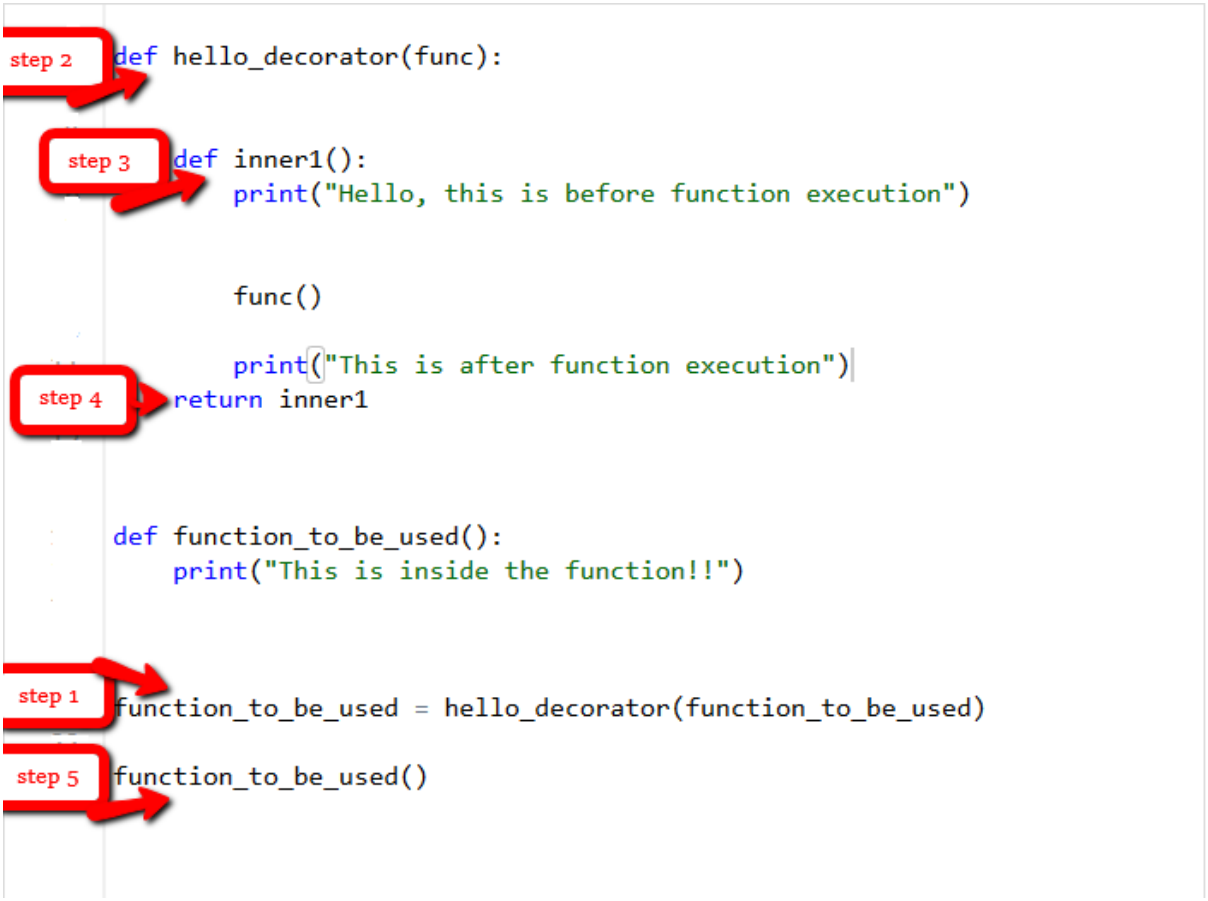
```
def hello_decorator(func):

    def inner1():

        print("Hello, this is before function
          execution")

    func()

        print("This is after function
          execution")

    return inner1



def xyz():

    print("This is inside the function !!")

def ppp():

    print(“hello”)

  # passing 'function_to_be_used' inside the
```

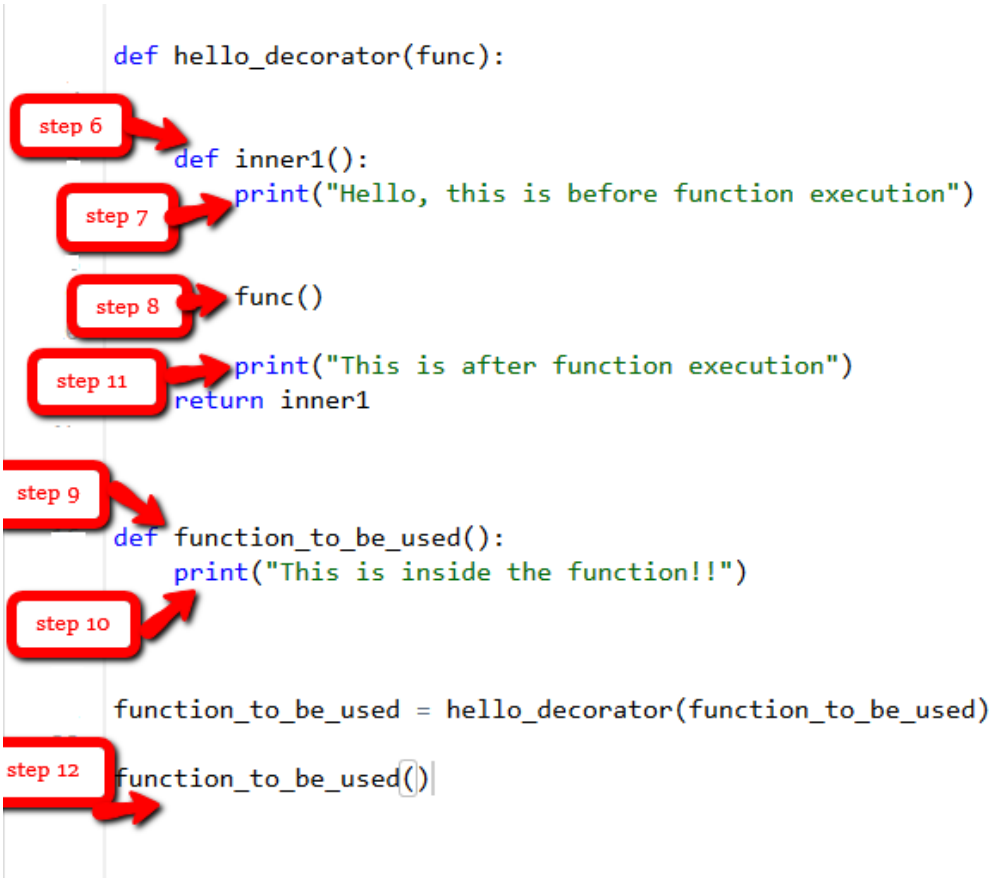
```
# decorator to control its behavior
xyz = hello_decorator(xyz)
# calling the function
xyz()
```

Output:

```
Hello, this is before function execution
This is inside the function !!
This is after function execution
```

Let's see the behavior of the above code how it runs step by step when the "function_to_be_used" is called.





Chaining Decorators
In simpler terms chaining decorators means decorating a function with multiple decorators.
Example:

```
# code for testing decorator chaining

def decor1(func):

    def inner():

        x = func()

        return x * x

    return inner



def decor(func):

    def inner():

        x = func()

        return 2 * x
```

```
    return inner


@decor


@decor1
def num():
    return 10


print(num())
```

Output:

```
400
```

The above example is similar to calling the function as –

```
decor(decor1(num))
```

Original Function to add two numbers.

```
def addTwoNumbers(a, b):
    c=a+b
    return c

c=addTwoNumber(4, 5)

print("Addition of two numbers=", c)
```

Output:

```
Addition of two numbers=9
```

Now our aim is to modify the behaviour of addTwoNumbers() without changing function definition and function call.

What function behaviour do we want to change?

We want addTwoNumbers function should calculate the sum of the square of two numbers instead of the sum of two numbers.

Here is a simple decorator to change the behaviour of the existing function.

```
def decorateFun(func):
```

```
        def sumOfSquare(x, y):
            return func(x**2, y**2)
        return sumOfSquare

@decorateFun
def addTwoNumbers(a, b):
    c = a+b
    return c

c = addTwoNumbers(4,5)
print("Addition of two numbers=", c)
```

Output:

```
Addition of two numbers=41
```

The below simple program is equivalent to the above decorator example. Here we are changing the function call.

```
def decorateFun(func):
    def sumOfSquare(x, y):
        return func(x**2, y**2)
    return sumOfSquare

def addTwoNumbers(a, b):
    c = a+b
    return c

obj=decorateFun(addTwoNumbers)
c=obj(4,5)
print("Addition of square of two numbers=", c)
```

Output:

```
Addition of square of two numbers=41
```

Note: The number of arguments to the function inside the decorators should be the same as the number of arguments to the actual function.