Python Decorators

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
```

Here, we have created the `inner()` function inside the `outer()` function.

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
```

Output

```
10
```

In the above example, the `calculate()` function takes a function as its argument. While calling `calculate()`, we are passing the `add()` function as the argument.

In the `calculate()` function, arguments: `func`, `x`, `y` become `add`, `4`, and `6` respectively.

And hence, `func(x, y)` becomes `add(4, 6)` which returns **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!
```

In the above example, the `return hello` statement returns the inner `hello()` function. This function is now assigned to the *greet* variable.

That's why, when we call `greet()` as a function, we get the output.

Python Decorators

As mentioned earlier, 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.

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


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

# Output: I am ordinary
```

Here, we have created two functions:

- `ordinary()` that prints `"I am ordinary"`
- `make_pretty()` that takes a function as its argument and has a nested function named `inner()`, and returns the inner function.

We are calling the `ordinary()` function normally, so we get the output `"I am ordinary"`. Now, let's call it using the decorator function.

```
def make_pretty(func):
    # define the inner function
    def inner():
        # add some additional behavior to decorated function
        print("I got decorated")

        # call original function
        func()
    # return the inner function
    return inner

# define ordinary function
def ordinary():
    print("I am ordinary")

# decorate the ordinary function
decorated_func = make_pretty(ordinary)

# call the decorated function
decorated_func()
```

Output

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

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

```
decorated_func = make_pretty(ordinary)
```

- We are now passing the `ordinary()` function as the argument to the `make_pretty()`.

- The `make_pretty()` function returns the inner function, and it is now assigned to the *decorated_func* variable.

```
decorated_func()
```

Here, we are actually calling the `inner()` function, where we are printing

@ 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,

```
def make_pretty(func):

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

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

ordinary()
```

Output

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

Here, the `ordinary()` function is decorated with the `make_pretty()` decorator using the `@make_pretty` syntax, which is equivalent to calling `ordinary = make_pretty(ordinary)`.

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:

```
def divide(a, b):
    return a/b
```

This function has two parameters, `a` and `b`. We know it will give an error if we pass in *b* as **0**.

Now let's make a decorator to check for this case that will cause the error.

```
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
```

Here, when we call the `divide()` function with the arguments **(2,5)**, the `inner()` function defined in the `smart_divide()` decorator is called instead.

This `inner()` function calls the original `divide()` function with the arguments **2** and **5** and returns the result, which is **0.4**.

Similarly, When we call the `divide()` function with the arguments **(2,0)**, the `inner()` function checks that `b` is equal to **0** and prints an error message before returning `None`.

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.

```
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

```
***************
%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%
***************
```

The above syntax of,

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

is equivalent to

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

The order in which we chain decorators matter. If we had reversed the order as,

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

The output would be:

```
%%%%%%%%%%%%%%%
***************
Hello
***************
%%%%%%%%%%%%%%%
```

Also Read:

- Python @property decorator
- Python property()

Table of Contents

- Introduction
- Prerequisites for learning decorators
- Python Decorators
- Decorating Functions with Parameters
- Chaining Decorators in Python