Python Decorators

programiz.com/python-programming/decorator

Join our newsletter for the latest updates.

A decorator takes in a function, adds some functionality and returns it. In this tutorial, you will learn how you can create a decorator and why you should use it.

Decorators in Python

Python has an interesting feature called **decorators** to add functionality to an existing code.

This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

Prerequisites for learning decorators

In order to understand about decorators, we must first know a few basic things in Python.

We must be comfortable with the fact that everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object.

Here is an example.

```
def first(msg):
    print(msg)


first("Hello")

second = first
second("Hello")
```

Output

```
Hello
Hello
```

When you run the code, both functions `first` and `second` give the same output. Here, the names `first` and `second` refer to the same function object.

Now things start getting weirder.

Functions can be passed as arguments to another function.

If you have used functions like `map`, `filter` and `reduce` in Python, then you already know about this.

Such functions that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.

```
def inc(x):
    return x + 1


def dec(x):
    return x - 1


def operate(func, x):
    result = func(x)
    return result
```

We invoke the function as follows.

```
>>> operate(inc,3)
4
>>> operate(dec,3)
2
```

Furthermore, a function can return another function.

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned


new = is_called()

# Outputs "Hello"
new()
```

Output

```
Hello
```

Here, `is_returned()` is a nested function which is defined and returned each time we call `is_called()`.

Finally, we must know about Closures in Python.

Getting back to Decorators

Functions and methods are called **callable** as they can be called.

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")
```

When you run the following codes in shell,

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

In the example shown above, `make_pretty()` is a decorator. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function `ordinary()` got decorated and the returned function was given the name `pretty`.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary).
```

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

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

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

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.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

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)
```

This new implementation will return `None` if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner, we can decorate functions that take parameters.

A keen observer will notice that parameters of the nested `inner()` function inside the decorator is the same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameters.

In Python, this magic is done as `function(*args, **kwargs)`. In this way, `args` will be the tuple of positional arguments and `kwargs` will be the dictionary of keyword arguments. An example of such a decorator will be:

```
def works_for_all(func):
    def inner(*args, **kwargs):
        print("I can decorate any function")
        return func(*args, **kwargs)
    return inner
```

Chaining Decorators in Python

Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    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
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```