Theory: Function decorators

28 minutes 0 / 5 problems solved

Decorator is a structural design pattern that allows programmers to extend and modify the behavior of a function, a method, or a class without changing their code. The main idea is that we place those callable objects, the functionality of which we need to change, inside other objects with new behavior. So, decorators are just wrappers around the initial objects. Most frequently, we use them to pass a function as an argument to a decorator to call this function later and perform some actions before and after the call.

§1. Syntax

In Python, the standard syntax for decorators is the `@` sign preceding the name of a decorator, and then the object we want to decorate on the next line with the same indentation. Decorators are called immediately before the body of a function, the behavior of which we would like to change. Here is a small example of how the general structure should look like:

```
@decorator_function
def func():
    ...
```

Now, to better understand how it works, let's see how to make a simple decorator.

```
def our_decorator(other_func):
    def wrapper(args_for_function):
        print('This happens before we call the function')
        return other_func(args_for_function)

    return wrapper
```

Here we define the function `our_decorator`, it takes another function as its argument and contains a `wrapper` that prints the message and calls the function that we have passed to `our_decorator`. Then, we return this wrapper function that contains our modified one.

Now, we define a function `greet` using `our_decorator`:

```
@our_decorator
def greet(name):
    print('Hello,', name)
```

Then, if we call `greet`, we will see the following output:

```
greet('Susie')
# This happens before we call the function
# Hello, Susie
```

However, you do not always need to write decorators, sometimes you can use decorators from the Python standard library.

§2. Why use decorators?

The reason why you may want to use decorators is that they provide means for making your code more readable and clean. Imagine that we have a set of functions. We want to measure, for instance, how long it takes for each of them to perform the algorithms, so we add timers in each code block:

Current topic:

Function decorators

Topic depends on:

✓ Declaring a function Stage 1 10

Topic is required for:

Abstract classes

Decorators in OOP

Table of contents:

↑ Function decorators

§1. Syntax

§2. Why use decorators?

§3. Summary

Feedback & Comments

```
import time

def func1(args_for_function):
    start = time.time()  # gets the current time
    ...                  # something happens here
    end = time.time()
    print('func1 takes', end - start, 'seconds')



def func2(args_for_function):

    start = time.time()

    ...

    end = time.time()

    print('func2 takes', end - start, 'seconds')
```

However, once it is done, the two following problems may arise:

- Particular lines would appear and be repeated in each function: the ones with `start` and `end` in our case;
- These lines would be redundant to the actual functionality and the initial code.

These issues can be solved with a separate reusable pattern that may be further applied to any other function. In our case, we can make it like this:

```
def timer(func):
    def wrapper(args_for_function):
        start = time.time()
        func(args_for_function)
        end = time.time()
        print('func takes', end - start, 'seconds')

    return wrapper




@timer

def func1(args_for_function):

    ...  # something happens here
```

In the example above, we have written a function decorator `timer()` that takes any function as an argument, it notes the time then invokes the function, notes the time again, and prints how much time it took. As a result, we can use this decorator for any function later on, and there will be no need to modify the code of the functions itself.

§3. Summary

Now, let's go over the main points we have learned in this topic:

- Decorators allow us to change the behavior of the object without changing its source code;
- They are introduced with the `@` symbol right before the function, the functionality of which we want to modify;
- To create custom decorators, we need to specify a decorator function that will return a wrapper over the given function.

Report a typo

145 users liked this theory. **23** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (7) Hints (1) Useful links (2) Show discussion