WORKING ON YOUR OWN

Day 29: Decorators

Welcome to day 29 of the [30 Days of Python] series! Today we're going to be learning about decorators.

This is a relatively advanced topic, and probably one of the trickiest things we've covered in this series. It's worth the struggle though, because decorators are an extremely powerful tool, and they feature a lot in professional Python code.

Many libraries also provide decorators for us to use, so we at least need to know what they are, and how to use them. Decorators are heavily used in web development with Flask, for example.

What is a decorator

Fundamentally, a decorator is just a function, or more generally a *callable*. We're going to focus on functions in this post, since they're the only callables we know about.

Decorators are special functions, because they take in some other function and they give us back a *new function* which can do something more than the function we passed in.

Decorators are very useful, because they allow us to very easily provide some additional functionality to many functions in our application. Due to their power and flexibility, they're used quite extensively in both the standard library modules, and third party libraries.

A quick review of functions

One thing we have to keep very clear in our minds when working with decorators is the difference between referencing a function, and calling a function.

Let's define a simple function to demonstrate the difference between these two concepts.

```
def add(a, b):
	return a + b
```

When we define a function like this, Python creates a function object which is a callable. In other words, we can the *call* the function using this object. This object is assigned to a variable with the same name as the function.

We've seen this before when looking at `globals()` in [day 13](day 13).

If we look at the global namespace after defining this `add` function, we see something like this:

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.SourceFileL
oader object at 0x7f2286f6c590>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)>,
        '__file__': 'app.py',
        '__cached__': None,
        'add': <function add at 0x7f2286f2be60>
}
```

As we can see, the namespace contains a variable called `add`, and the value for this variable is a function object with the name `add`. This function object contains all the information Python needs to run our `add` function.

Whenever we write `add`, we're just referencing this function object. This is just a value like any other, since functions are first class citizens in the Python language.

This is very different from when we write `add()`. Here we're first referencing the function to get hold of the function object, and then the parentheses tell Python we want to *call* the function we just referenced.

`add` and `add()` are both expressions, but the first evaluates to a function object, and the second evaluates to whatever the function returns when we call it.

Defining a simple decorator

Let's start with an example which we can deconstruct in a moment.

```python
from typing import Callable

def example_decorator(func: Callable) -> Callable:
        def inner():
                print(f"Now calling {func.__name__}...")
                func()
                print(f"{func.__name__} has ended.")

        return inner
```

Style note

I've imported the `typing` module here so that we can better see what is going into this `example_decorator` function, and what is being returned by it. As we can see, in both cases it's a `Callable`, or more specifically, a function.

We don't need the `typing` module to use decorators.

So, what is going on in this `example_decorator`?

Starting with the top line, we can see that we have a single parameter in this case, which called `func`. As we've already discussed, we expect the argument passed to `func` to be a function. So far, so good.

Now we get to the function body of `example_decorator`, and things get interesting. Inside the function body, we've defined a new function. This new function is called `inner`, and just to be clear, the name is not special.

This `inner` function does a few things when it gets called.

First it prints a line to the console, then it calls `func`, which is an alias for the function we passed into `example_decorator`, and then it prints another line to the console. In case you're wondering, the `.__name__` is just an attribute which we can use to find the name of a function.

Note however, that we don't actually call `inner`. We've just defined this new function.

After we're done defining `inner`, we return a reference to `inner` from `example_decorator`.

Let's see this decorator in action to get a better idea of what's going on. To use the decorator, we can do something like this:

```
from typing import Callable

def example_decorator(func: Callable) -> Callable:
        def inner():
                print(f"Now calling {func.__name__}...")
                func()
                print(f"{func.__name__} has ended.")

        return inner

def greeter():
    print("Hello!")

greeter = example_decorator(greeter)
greeter()
```

Here we've defined another function called `greeter`, which just prints `"Hello!"` to the console when we call it.

All the real action goes on on this line:

```
greeter = example_decorator(greeter)
```

Here're we've called our `example_decorator` function, and we've passed in a *reference* to `greeter`. We've then assigned whatever the return value of `example_decorator` was to `greeter`, replacing the reference to the old `greeter` function.

In other words, the name `greeter` now refers to something completely different.

When we call `greeter` now, we get this:

```
Now calling greeter...
Hello!
greeter has ended.
```

And if we print `greeter` to find out its value, we see that it no longer refers to the old `greeter` function.

```
<function example_decorator.<locals>.inner at 0x7f6e06326830>
```

This makes a lot of sense, because we assigned the return value of `example_decorator` to `greeter`, and the return value of `example_decorator` was a reference to `inner`.

That means when we use the name `greeter`, we're now talking about the `inner` function, which explains the output we saw. We got exactly what we defined in `inner`.

We got a line printed to the console; we called `func`, which was the old `greeter` function in this case; and then we printed another line to console.

The `@` syntax

Before we look at some more decorator examples, let's talk about a piece of syntax that makes using decorators a lot easier.

In the last example, we had to do this:

```
greeter = example_decorator(greeter)
```

That is really clunky, in my opinion, and thankfully the Python authors gave a much nicer way of doing this kind of operation.

Instead of doing this assignment, we can decorate a function by putting `@` followed by the decorator name. This is placed right above the function we want to decorate.

This is absolutely identical to what we did before:

```python
from typing import Callable

def example_decorator(func: Callable) -> Callable:
        def inner():
                print(f"Now calling {func.__name__}...")
                func()
                print(f"{func.__name__} has ended.")

        return inner

@example_decorator
def greeter():
    print("Hello!")

greeter()
```

Decorating functions with arguments

So far we've decorated a function that doesn't take any arguments, but how do we decorate a function like this?

```python
def add(a, b):
        print(a + b)
```

If we try our old method, we're going to end up with a `TypeError`

If we try our old method, we're going to end up with a `TypeError`, because when we call `func` in the inner function, we've been calling it without arguments.

In order to solve this, we need to define a set of parameters for our `inner` function, and we can pass the arguments on when we call `func`.

One problem we have to overcome here is that our decorators are meant to be quite general, so that we can reuse them with many different functions. We therefore shouldn't do something like this:

```
from typing import Callable, Union

Real = Union[int, float]

def calculate(func: Callable) -> Callable:
	def inner(a, b):
		print("Calculating...")
		func(a, b)

	return inner

@calculate
def add(a: Real, b: Real):
	print(a + b)

add(1, 5)
```

It works, but it only works when the function has two parameters. If we want to perform a unary operation, or we want to be able to decorate something like `sum`, we'd need a different decorator.

Luckily we've already learnt about some perfect tools for this job: `*args` and `**kwargs`.

Using `*args` and `**kwargs` we can accept any set of arguments we like for `inner`, and then we can pass them right along to the `func` like for `inner`, and then we can pass them right along to the `func`.

```python
from typing import Callable, Union

Real = Union[int, float]

def calculate(func: Callable) -> Callable:
	def inner(*args, **kwargs):
		print("Calculating...")
		func(*args, **kwargs)

	return inner

@calculate
def add(a: Real, b: Real):
	print(a + b)

add(1, 5)
```

If you need a refresher on `*args` and `**kwargs`, have another look at [day 17](#).

Decorating functions with return values

Now that we've tackled functions with arguments, we now have another problem. What if the function we're decorating returns a value?

At the moment, we'll just return it inside of `inner`, but we don't do anything with this value. We don't pass that value on. Luckily this is a very easy one to solve: we just have to return the value from `inner` as well.

For example, here is a very silly decorator which gives us the wrong answer for our calculations:

```python
from typing import Callable, Union
```

```python
Real = Union[int, float]

def wrong_answer(func: Callable) -> Callable:
        def inner(*args, **kwargs):
                return func(*args, **kwargs) + 1

        return inner

@wrong_answer
def add(a: Real, b: Real) -> Real:
        return a + b

print(add(1, 5))  # 7
```

The `wraps` function

One final thing I want to talk about in this post is a function in the standard library called `wraps` , which is in the `functools` module. We use `wraps` to decorate our `inner` functions, and its job is to preserve the original name of the function we passed into the decorator.

If you remember back to our original example, referencing the variable name `greeter` gave us back something like this:

```
<function example_decorator.<locals>.inner at 0x7f6e06326830>
```

The name of our original `greeter` function is totally gone.

Using `wraps` , we can preserve the original function name, and importantly, any documentation included in that function.

`wraps` is not strictly speaking a decorator: it's a decorator factory. That means it's a function that returns a decorator. This isn't a detail we need to worry about, but it does mean the syntax is slightly different, because `wraps` is able to take its own arguments.

When we call `wraps` , we use the same @ syntax, and we pass in `func`

When we call `wraps`, we use the same `@` syntax, and we pass in `func` as an argument. Here is an example:

```python
from functools import wraps
from typing import Callable

def example_decorator(func: Callable) -> Callable:
        @wraps(func)
        def inner():
                print(f"Now calling {func.__name__}...")
                func()
                print(f"{func.__name__} has ended.")

        return inner

def greeter():
    print("Hello!")

greeter = example_decorator(greeter)
print(greeter)  # <function greeter at 0x7fcce99a8830>
```

As you can see, the function we got back from `example_decorator` is still called `greeter`, despite us actually returning the `inner` function.

A real world example

Now that we've learnt how to decorate various types of functions, let's look at an example which is a bit more practical. We're going to create a decorator that will time our code for us.

This is a handy thing to make, because it allows us to test several implementations to see which option is faster.

In order to actually test our code, we're going to use the `time` module, which contains a function called `perf_counter`. `perf_counter` is going to give us a very detailed representation of the current time.

By calling `perf_counter`, running our code, and calling `perf_counter` again, we can get a good idea of the time that elapsed between the again, we can get a good idea of the time that elapsed between the two called to `perf_counter` .

Here is one possible implementation:

```
from functools import wraps
from time import perf_counter
from typing import Callable

def stopwatch(func: Callable) -> Callable:
        @wraps(func)
        def inner(*args, **kwargs):
                start_time = perf_counter()
                func(*args, **kwargs)
                stop_time = perf_counter()

                print(f"{func.__name__} ran in {stop_time - s
tart_time:.5f}s")

        return inner
```

Now let's use this to compare some code. I'm going to compare how long it takes for Python to make a list from a 100,000 number range, and how long it takes to do the same for a set.

```
from functools import wraps
from time import perf_counter
from typing import Callable, List, Set

def stopwatch(func: Callable) -> Callable:
        @wraps(func)
        def inner(*args, **kwargs):
                start_time = perf_counter()
                func(*args, **kwargs)
                stop_time = perf_counter()

                print(f"{func.__name__} ran in {stop_time - s
tart_time:.5f}s")
```

```
        return inner

@stopwatch
def make_list(size: int) -> List:
    return list(range(size))

@stopwatch
def make_set(size: int) -> Set:
    return set(range(size))

make_list(100_000)  # make_list ran in 0.00407s
make_set(100_000)   # make_set ran in 0.00628s
```

If you run this test numerous times, you may find some significant fluctuation in the time these operations take. That's perfectly normal, and should be expected, since your computer is running all kinds of other processes at the same time.

It would probably be better if our performance counter ran the test a certain number of times, and computed an average time instead.

The best way to do this would be using a decorator factory, which would allow us to pass in arguments to our decorator, but using our current knowledge, we can at least have the test run 10 times.

```
from functools import wraps
from math import fsum
from time import perf_counter
from typing import Callable, List, Set

def stopwatch(func: Callable) -> Callable:
        @wraps(func)
        def inner(*args, **kwargs):
                times = []

                for _ in range(10):
                        start_time = perf_counter()
                        func(*args, **kwargs)
```

```
                                stop_time = perf_counter()

                                elapsed = stop_time - start_time
                                times.append(elapsed)

                        average_time = fsum(times) / len(times)

                        print(f"{func.__name__} ran in {average_time:
.5f}s on average")

                return inner

@stopwatch
def make_list(size: int) -> List:
    return list(range(size))

@stopwatch
def make_set(size: int) -> Set:
    return set(range(size))

make_list(100_000)  # make_list ran in 0.00337s on average
make_set(100_000)   # make_set ran in 0.00565s on average
```

Now our performance tests should see a lot less fluctuation, giving us a better picture of the performance differences.

Exercises

Make a decorator which calls a given function twice. You can assume the functions don't return anything important, but they may take arguments.

Imagine you have a list called `books`, which several functions in your application interact with. Write a decorator which causes your functions to run only if `books` is not empty.

Write a decorator called `printer` which causes any decorated function to print their return values. If the return value of a given function is

`None` , `printer` should do nothing.

You can find our solution here.

Additional Resources

We actually have an entire series on decorators that you can check out on our YouTube channel to get more information and to see more examples.

Also, if you're looking for a more robust option for testing your code, you should check out the `timeit` module. You can find documentation for `timeit` here.
