06. Decorators

October 15, 2025

Decorators modify or introduce code in methods and functions dynamically. A decorator is basically a function that receives another function as parameter, adds/modifies its functionality and returns it.

0.0.1 1. Functions roles and uses

In order to understand decorators, we must first remember a few things about functions.

Various names can be bound to the same function object:

[43]:
```
def greet(name):
    print(f'Hello, {name}!')

greet('John')

salute = greet
salute('John')

print(id(salute) == id(greet))
print(salute is greet)
```

```
Hello, John!
Hello, John!
True
True
```

Functions can be passed as arguments to another function:

[44]:
```
def add(a, b):
    return a + b

def diff(a, b):
    return a - b

def compute(a, b, operation):
    return operation(a, b)


print(compute(10, 2, add))
print(compute(10, 2, diff))
```

```
print(compute(10, 2, pow))
print(compute(10, 2, lambda x, y: x / y))
```

```
12
8
100
5.0
```

Functions can be defined inside other functions:

[45]:
```
def func():
    def inner_func():
        print('Inside!')
    inner_func()

func()
```

```
Inside!
```

A function can return another function:

[46]:
```
def func():
    def inner_func():
        print('Inside!')
    return inner_func

func_returned = func()
print(func_returned)
print(type(func_returned))
```

```
<function func.<locals>.inner_func at 0x105c30180>
<class 'function'>
```

[47]:
```
func_returned()
```

```
Inside!
```

0.0.2 2. Simple decorators

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

In fact, any object which implements the special method `__call__()` is a 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.

[48]:
```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
```

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


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`.

Generally, we decorate a function and reassign it to its initial name:

```
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 decorated it.

[49]:
```
@make_pretty
def ordinary():
    print("I am ordinary")
```

The example above is equivalent to:

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

ordinary = make_pretty(ordinary)
```

The `@decorator` notation is just syntactic sugar.

Generally, decorators should be able to decorate any function. Let's see what happens when we try do decorate a function that receives an argument:

[50]:
```
@make_pretty
def greet(name):
    print(f'Hello, {name}!')

try:
    greet('Anna')
except Exception as ex:
    print(f'{type(ex).__name__}: {ex}')
```

```
TypeError: make_pretty.<locals>.inner() takes 0 positional arguments but 1 was
given
```

It looks like our decorator isn't general enough. Because `make_pretty()` returns `inner` function, we should change `inner` to accept any number of parameters and pass them along to the `func()` call inside `inner()`.

[51]:
```
def make_pretty(func):
    def inner(*args, **kwargs):
        print("I got decorated")
        func(*args, **kwargs)
    return inner
```

Let's try decorating `greet` again:

[52]:
```
@make_pretty
def greet(name):
    print(f'Hello, {name}!')

greet('Anna')
```

```
I got decorated
Hello, Anna!
```

There is still one detail we have left out. Let's try decorating a function that returns some value:

[53]:
```
@make_pretty
def increment(num, step=1):
    return num + step

result = increment(100)
print('Incremented value:', result)
```

```
I got decorated
Incremented value: None
```

The incremented value should be `101`, but instead, it's `None`. Let's take another look at the `inner` function: it simply calls `func`, but ignores the value returned by it.

[54]:
```
def make_pretty(func):
    def inner(*args, **kwargs):
        print("I got decorated")
        return func(*args, **kwargs)
    return inner

@make_pretty
def increment(num, step=1):
    """Increments num with step. If not provided, step=1"""
    return num + step

result = increment(100, 2)
print('Incremented value:', result)
```

```
I got decorated
Incremented value: 102
```

This time, our decorator seems to work properly. Let's check one last detail:

[55]:
```
print(increment, increment.__doc__)
```

```
<function make_pretty.<locals>.inner at 0x105c311c0> None
```

When inspecting `increment` function, we can see it points to the `make_pretty.<locals>.inner` function object and it has lost its properties, like the docstring. In order to prevent this from happening, we can use `functools.wraps` decorator:

[56]:
```
import functools

def make_pretty(func):
    @functools.wraps(func)
    def inner(*args, **kwargs):
        print("I got decorated")
        return func(*args, **kwargs)
    return inner

@make_pretty
def increment(num, step=1):
    """Increments num with step. If not provided, step=1"""
    return num + step

result = increment(100, 2)
print('Incremented value:', result)
```

```
I got decorated
Incremented value: 102
```

[57]:
```
print(increment, increment.__doc__)
```

```
<function increment at 0x105be9580> Increments num with step. If not provided,
step=1
```

0.0.3 Exercises 2

1. Write a decorator that computes (and displays) execution time for a function. Use `time.sleep` or some nested loops with large iteration number to simulate slow functions.
2. Write a decorator function and prints the number of times the decorated function has been called. Decorate at least two different functions with it and call each function multiple times.

0.0.4 3. Decorators with parameters

To create a decorator that takes parameters, you need an additional layer of function nesting. Here's the structure:

1. **The outer function**: This takes the parameters you want to pass to the decorator.

2. **The decorator function**: This takes the function to be decorated.
3. **The wrapper function**: This wraps the original function, adding the desired behavior.

Here's an example of a decorator with parameters:

[58]:
```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat
```

We will decorate the function `greet`:

[59]:
```
@repeat(num_times=3)
def greet(name):
    print(f"Hello, {name}!")
```

Remember that decoration is syntactic sugar for:

`greet = repeat(num_times=3)(greet)`

So, basically, we're using the function returned by `repeat` (`decorator_repeat`) as a simple decorator for `greet`. When inspecting `greet`, we see that it is actually `wrapper`:

[60]:
```
greet
```

[60]:
```
<function __main__.repeat.<locals>.decorator_repeat.<locals>.wrapper(*args,
**kwargs)>
```

[61]:
```
greet("Anna")
```

```
Hello, Anna!
Hello, Anna!
Hello, Anna!
```

0.0.5 Exercises 3

1. Write a decorator `retry` that accepts a parameter `num_times`. The decorator will modify the behavior of the decorated function:
   - if an exception occurrs in the decorated function, it will not be automatically raised, the function will retry `num_times` times
   - if the function fails in all tries, the last occurring exception should still be raised

   A decorator like this one would be useful in a scenario where a certain resource isn't available at all times (imagine a server timing out). To emulate such a behavior, you can use a function like the one below: `python  def divide():      x = int(input("Enter first number: "))      y = int(input("Enter second number: "))      return x / y`