14. Decorators

September 19, 2025

1 Decorators

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.

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:

[16]:
```python
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:

[17]:
```python
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:

[18]:
```
def func():
    def inner_func():
        print('Inside!')
    inner_func()

func()
```

```
Inside!
```

A function can return another function:

[19]:
```
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 0x1052ee200>
<class 'function'>
```

[20]:
```
func_returned()
```

```
Inside!
```

1.2 Simple decorators

Functions and methods are called **callable** as 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.

[21]:
```
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.

[22]:
```
@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:

[23]:
```
@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()`.

[24]:
```
def make_pretty(func):
    def inner(*args, **kwargs):
        print("I got decorated")
        func(*args, **kwargs)
    return inner
```

Let's try decorating `greet` again:

[25]:
```
@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:

[26]:
```
@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.

[27]:
```
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:

[28]:
```
print(increment, increment.__doc__)
```

```
<function make_pretty.<locals>.inner at 0x1052edc60> 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:

[29]:
```
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
```

[30]:
```
print(increment, increment.__doc__)
```

```
<function increment at 0x1052ed6c0> Increments num with step. If not provided,
step=1
```

1.3 Exercises

1. Write a decorator that computes (and displays) execution time for a function. Hint: `time.time()` function returns current time in seconds.
2. Write a decorator function that takes a function as an argument and prints the number of times the function has been called.