18. Decorators

By definition, a **decorator** *is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.*

Put simply: **decorators wrap a function, modifying its behavior**. But it **modifies not a function call, but a functon's definition!**

Basic syntax:

```python
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator

@decorator
def func():  # decorated function
    ...
```

1 Functions revisited

Before understanding decorators, understanding of how functions work is needed.

While not a purely functional language, Python supports many of the *functional programming concepts*, including **functions as first-class objects**.

1.1 Functions are first-class objects

This means that **functions can be passed around and used as arguments**, just like any other object (string, int, float, list, etc.):

```python
def say_hello(name):
    return f'Hello, {name}'

def greet_bob(func): # this function expects another function as argument
    return func('Bob')

greet_bob(say_hello)
> 'Hello, Bob'
```

1.2 Inner functions

It is possible to define **functions inside other functions** – **inner functions**.

Example:

```python
def parent():
    print("Printing from the parent() function")

    def first_child():
        print("Printing from the first_child() function")
    def second_child():
        print("Printing from the second_child() function")

    second_child()
    first_child()

parent()
> Printing from the parent() function
> Printing from the second_child() function
> Printing from the first_child() function
```

Furthermore, the inner functions are not defined until the parent function is called. Whenever you call parent function, the inner functions are also called. But because of their local scope, they **are not available outside of the parent function**.

1.3 Returning functions from functions

Example:

```
def parent(num):
    def first_child():
        return "Hi, I am Emma"
    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child # note that you return func name w/o parentheses
    else:
        return second_child
```

Note that **you return function without parentheses** i.e. this means that **you are returning the *reference to the function***. In contrast function **with parentheses refers to the result of evaluating the function**:

```
parent      —> <function __main__.parent(num)>
parent(1)   —> <function __main__.parent.<locals>.first_child()>
parent(1)() —> 'Hi, I am Emma'
```

2 Simple decorators

2.1 Introductory examples

Example 1

```
# Decorator definition:
def my_decorator(func):
    def wrapper():
        print('smth happened before')
        func()
        print('smth happened after')
    return wrapper

# Function to be modified:
def say_whee():
    print('   Whee!')

# Wrap, decoration:
say_whee = my_decorator(say_whee)
```

```
say_whee()
> smth happened before
> Whee!
> smth happened after
```

In effect when you wrapped and made decoration in the last line, **original function name** `say_whee()` **now points to the** `wrapper()` **inner function**:

```
say_whee
> <function __main__.my_decorator.<locals>.wrapper()>
```

Note: **You can name your inner function whatever you want**, and a generic name like `wrapper()` is usually okay. Next, **we will name the inner function with the same name as the decorator but with a** `wrapper_` **prefix**.

Example 2

Because `wrapper()` is a regular Python function, the way a decorator modifies a function can change dynamically i.e. **we can replace decorators**!

The following example will only run the decorated code during the day:

```
from datetime import datetime

def not_at_night(func):
    def wrapper():
        if 9 <= datetime.now().hour < 23:
            func()
        else:
             pass
    return wrapper

def say_whee():
    print('Whee!')

say_whee = not_at_night(say_whee)
```

2.2 ”Pie” syntax

Syntax by example:

```
def my_decorator(func):
    def wrapper():
        print('smth happened before')
        func()
        print('smth happened after')
    return wrapper

@my_decorator
def say_whee():
    print('Whee!')
```

```
say_whee()
> smth happened before
> Whee!
> smth happened after
```

2.3 Reusing decorators

Decorator is just a regular Python function. All the usual tools for easy **reusability are available**.

Create a file called `decorators.py` with the following content:

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

You can now use this new decorator in other files **by doing a regular import**:

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")

say_whee()
> Whee!
> Whee!
```

2.4 Decorating functions with arguments

Say, we need to decorate a function that accepts some arguments:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f'Hello, {name}')

greet('World')
> ERROR
```

The problem is that the inner function `wrapper_do_twice()` does not take any arguments, but `name='World'` was passed to it. You could fix this by letting `wrapper_do_twice()` accept one argument, but then it would not work for the `say_whee()` function you created earlier.

Solution: is to **use** `*args` **and** `**kwargs` **in the inner wrapper function definition and calls**. Then it will accept an arbitrary number of positional and keyword arguments.

Rewrite decorators.py as follows:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

```
greet('World')
> Hello World
> Hello World
```

2.5 Returning values from decorated functions

What happens to the return value of decorated functions?

```
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"

return_greeting("Adam")
> Creating greeting
> Creating greeting
```

Decorator ate the return value from the function. Because the `do_twice_wrapper()` doesn't explicitly return a value, the call `return_greeting("Adam")` ended up returning `None`.

Solution: make wrapper function returning the return value of the decorated function. Change your `decorators.py` file:

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs): # MODIFIED - accept args of wrapped function
        func(*args, **kwargs)
        value = func(*args, **kwargs)  # REPLACED
        return value                   # REPLACED
    return wrapper_do_twice

return_greeting("Adam")
> Creating greeting
> Creating greeting
> 'Hi Adam'
```

2.6 '@functools.wraps'

After being decorated, the original function looses its identity and uses that of the wrapper!

Solution: decorators should **use the `@functools.wraps` decorator**, **which preserves information about the original function.**

Update `decorators.py` again:

```
import functools  # ADDED

def do_twice(func):
    @functools.wraps(func)  # ADDED
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice

greet('Paul')
> Hello Paul
> Hello Paul
```

Now:

```
greet --> <function __main__.greet(name)>

greet.__name__ --> 'greet'

help(greet) -->
Help on function greet in module __main__:
greet(name)
```

while previously it was (the identity of the decorator):

```
greet --> <function decorators.do_twice.<locals>.wrapper_do_twice(*args, **kwargs)>

greet.__name__ --> 'wrapper_do_twice'

help(greet) -->
Help on function wrapper_do_twice in module decorators:
wrapper_do_twice(*args, **kwargs)
```

3 Real-world use-cases

3.1 Timing the function

Create a `@timer` decorator: **measure the time a function takes to execute** and print the duration to the console.

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1

        value = func(*args, **kwargs)

        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} s")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
    sum([i**2 for i in range(10000)])
    return "Done!"
```

```
waste_some_time(100)
> Finished 'waste_some_time' in 0.4642 s
> 'Done!'
```

Note: The `@timer` decorator is great if you just want to get an idea about the runtime of your functions. If you want to do more precise measurements of code, **you should instead consider the `timeit` module in the standard library.** It temporarily disables garbage collection and runs multiple trials to strip out noise from quick function calls.

3.2 Debugging code

`@debug` decorator will **print the arguments a function is called with as well as its return value** every time the function is called:

```
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]                       # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)            # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)

        print(f"{func.__name__!r} returned {value!r}")            # 4
        return value
    return wrapper_debug


@debug
def make_greeting(name, age=None):
    if age is None:
       return f"Howdy {name}!"
    else:
        return f"Whoa {name}! {age} already, you are growing up!"
```

```
make_greeting('Andy', '25')
> Calling make_greeting('Andy', '25')
> 'make_greeting' returned 'Whoa Andy! 25 already, you are growing up!'
> 'Whoa Andy! 25 already, you are growing up!'
```