**Python Decorators That Can Reduce Your Code By Half**

[[Ayush Thakur](https://medium.com/@ayush-thakur02?source=post_page-----b19f673bc7d8--------------------------------)](https://medium.com/@ayush-thakur02?source=post_page-----b19f673bc7d8--------------------------------)

[Ayush Thakur](https://medium.com/@ayush-thakur02?source=post_page-----b19f673bc7d8--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fc61c106e0290&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40ayush-thakur02%2Fpython-decorators-that-can-reduce-your-code-by-half-b19f673bc7d8&user=Ayush+Thakur&userId=c61c106e0290&source=post_page-c61c106e0290----b19f673bc7d8---------------------post_header-----------)

5 min read

·

Nov 14

1.7K

19

Hi there, welcome to my blog! Today I’m going to share with you some amazing Python decorators that can reduce your code by half. Sounds too good to be true, right? Well, let me show you how they work and why you should use them in your projects.

**What are Python decorators?**

Python decorators are a powerful feature that allow you to modify the behavior of a function or a class without changing its source code. They are essentially functions that take another function as an argument and return a new function that wraps the original one. This way, you can add some extra functionality or logic to the original function without modifying it.

For example, suppose you have a function that prints a message to the console:

def hello():  
 print("Hello, world!")

Now, suppose you want to measure how long it takes to execute this function. You could write another function that uses the time module to calculate the execution time and then calls the original function:

import time  
  
def measure\_time(func):  
 def wrapper():  
 start = time.time()  
 func()  
 end = time.time()  
 print(f"Execution time: {end - start} seconds")  
 return wrapper

Notice that the measure\_time function returns another function called wrapper, which is the modified version of the original function. The wrapper function does two things: it records the start and end time of the execution, and it calls the original function.

Now, to use this function, you could do something like this:

hello = measure\_time(hello)  
hello()

This would output something like this:

Hello, world!  
Execution time: 0.000123456789 seconds

As you can see, we have successfully added some extra functionality to the hello function without changing its code. However, there is a more elegant and concise way to do this using decorators. Decorators are simply syntactic sugar that allow you to apply a function to another function using the @ symbol. For example, we could rewrite the previous code like this:

@measure\_time  
def hello():  
 print("Hello, world!")  
  
hello()

This would produce the same output as before, but with much less code. The @measure\_time line is equivalent to saying hello = measure\_time(hello), but it looks much cleaner and more readable.

**Why use Python decorators?**

Python decorators are useful for many reasons, such as:

* They allow you to reuse code and avoid repetition. For example, if you have many functions that need to measure their execution time, you can simply apply the same decorator to all of them instead of writing the same code over and over again.
* They allow you to separate concerns and follow the principle of single responsibility. For example, if you have a function that performs some complex calculation, you can use a decorator to handle the logging, error handling, caching, or validation of the input and output, without cluttering the main logic of the function.
* They allow you to extend the functionality of existing functions or classes without modifying their source code. For example, if you are using a third-party library that provides some useful functions or classes, but you want to add some extra features or behavior to them, you can use decorators to wrap them and customize them to your needs.

**Some examples of Python decorators**

There are many built-in decorators in Python, such as @staticmethod, @classmethod, @property, @functools.lru\_cache, @functools.singledispatch, etc. You can also create your own custom decorators for various purposes. Here are some examples of Python decorators that can reduce your code by half:

**1. The**@timer**decorator**

This decorator is similar to the @measure\_time decorator we saw before, but it can be applied to any function that takes any number of arguments and returns any value. It also uses the functools.wraps decorator to preserve the name and docstring of the original function. Here is the code:

import time  
from functools import wraps  
  
def timer(func):  
 @wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 start = time.time()  
 result = func(\*args, \*\*kwargs)  
 end = time.time()  
 print(f"Execution time of {func.\_\_name\_\_}: {end - start} seconds")  
 return result  
 return wrapper

Now, you can use this decorator to measure the execution time of any function, such as:

@timer  
def factorial(n):  
 """Returns the factorial of n"""  
 if n == 0 or n == 1:  
 return 1  
 else:  
 return n \* factorial(n - 1)  
  
@timer  
def fibonacci(n):  
 """Returns the nth Fibonacci number"""  
 if n == 0 or n == 1:  
 return n  
 else:  
 return fibonacci(n - 1) + fibonacci(n - 2)  
  
print(factorial(10))  
print(fibonacci(10))

This would output something like this:

Execution time of factorial: 1.1920928955078125e-06 seconds  
3628800  
Execution time of fibonacci: 0.000123456789 seconds  
55

**2. The**@debug**decorator**

This decorator is useful for debugging purposes, as it prints the name, arguments, and return value of the function it wraps. It also uses the functools.wraps decorator to preserve the name and docstring of the original function. Here is the code:

from functools import wraps  
  
def debug(func):  
 @wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Calling {func.\_\_name\_\_} with args: {args} and kwargs: {kwargs}")  
 result = func(\*args, \*\*kwargs)  
 print(f"{func.\_\_name\_\_} returned: {result}")  
 return result  
 return wrapper

Now, you can use this decorator to debug any function, such as:

@debug  
def add(x, y):  
 """Returns the sum of x and y"""  
 return x + y  
  
@debug  
def greet(name, message="Hello"):  
 """Returns a greeting message with the name"""  
 return f"{message}, {name}!"  
  
print(add(2, 3))  
print(greet("Alice"))  
print(greet("Bob", message="Hi"))

This would output something like this:

Calling add with args: (2, 3) and kwargs: {}  
add returned: 5  
5  
Calling greet with args: ('Alice',) and kwargs: {}  
greet returned: Hello, Alice!  
Hello, Alice!  
Calling greet with args: ('Bob',) and kwargs: {'message': 'Hi'}  
greet returned: Hi, Bob!  
Hi, Bob!

**3. The**@memoize**decorator**

This decorator is useful for optimizing the performance of recursive or expensive functions, as it caches the results of previous calls and returns them if the same arguments are passed again. It also uses the functools.wraps decorator to preserve the name and docstring of the original function. Here is the code:

from functools import wraps  
  
def memoize(func):  
 cache = {}  
 @wraps(func)  
 def wrapper(\*args):  
 if args in cache:  
 return cache[args]  
 else:  
 result = func(\*args)  
 cache[args] = result  
 return result  
 return wrapper

Now, you can use this decorator to memoize any function, such as:

@memoize  
def factorial(n):  
 """Returns the factorial of n"""  
 if n == 0 or n == 1:  
 return 1  
 else:  
 return n \* factorial(n - 1)  
@memoize  
def fibonacci(n):  
 """Returns the nth Fibonacci number"""  
 if n == 0 or n == 1:  
 return n  
 else:  
 return fibonacci(n - 1) + fibonacci(n - 2)  
print(factorial(10))  
print(fibonacci(10))

This would output the same as before, but with much faster execution time, as the results are cached and reused.

**Conclusion**

Python decorators are a powerful and elegant way to modify the behavior of functions or classes without changing their source code. They can help you reduce your code by half, improve your code readability, reuse your code, separate your concerns, and extend the functionality of existing code. I hope you enjoyed this blog post and learned something new. If you have any questions or comments, feel free to leave them below. And don’t forget to share this post with your friends and colleagues who might be interested in learning more about Python decorators. Thanks for reading!