

Understanding Python Decorators



Alexander Obregon · [Follow](#)

11 min read · May 29, 2024



11



1



[Image Source](#)

Introduction

Python decorators are a powerful and flexible feature that allows you to modify the behavior of functions or methods without changing their actual

code. They provide a clean and readable way to extend the functionality of existing functions and are widely used in Python programming for logging, access control, instrumentation, and more. This article will what into what decorators are, how they work, and how you can create and use them effectively in your Python code.

What are Python Decorators?

A decorator in Python is a function that takes another function and extends its behavior without explicitly modifying it. Decorators are often used to add “wrapping” functionality to existing code in a concise and reusable manner. They play an important role in many Python frameworks and libraries, enabling developers to write more modular, maintainable, and readable code.

The Basics of Decorators

In Python, decorators are often used with the `@` symbol followed by the decorator function name, placed above the function to be decorated. This syntactic sugar makes it clear and straightforward to apply a decorator. Here's a simple example to illustrate this:

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
@my_decorator  
def say_hello():  
    print("Hello!")  
  
say_hello()
```

In this example, `my_decorator` is a decorator that wraps the `say_hello` function, adding behavior before and after the call to `say_hello`. When `say_hello` is called, it first prints a message, then calls the original `say_hello` function, and finally prints another message. The `@my_decorator` syntax is a shorthand for writing `say_hello = my_decorator(say_hello)`.

How Decorators Work

To understand how decorators work, it is important to grasp the concept of higher-order functions. A higher-order function is a function that takes another function as an argument or returns a function as a result. Decorators leverage this concept by taking a function, adding some functionality, and returning the enhanced function.

When you use the `@my_decorator` syntax, Python performs the following steps:

1. It takes the function `say_hello`.
2. Passes it to the `my_decorator` function.
3. Assigns the result (the `wrapper` function) back to `say_hello`.

Thus, when you call `say_hello`, you are actually calling the `wrapper` function, which includes the original `say_hello` functionality along with the added behavior.

Functions as First-Class Objects

In Python, functions are first-class objects. This means they can be passed around and used as arguments, just like any other object (e.g., strings, integers, lists). This feature is fundamental to the concept of decorators. Here's an example to demonstrate functions as first-class objects:

```
def greet(name):  
    return f"Hello, {name}!"  
  
def call_func(func, value):  
    return func(value)  
  
print(call_func(greet, "World"))
```

In this example, the `call_func` function takes another function `func` and a value `value` as arguments. It calls `func` with `value` and returns the result. This flexibility allows decorators to modify the behavior of functions dynamically.

Nesting Functions

Decorators often use nested functions (functions defined within other functions) to wrap the original function. The inner function (wrapper) can perform actions before and after calling the original function. Here's an extended example:

```
def my_decorator(func):  
    def wrapper(*args, **kwargs):  
        print("Something is happening before the function is called.")  
        result = func(*args, **kwargs)  
        print("Something is happening after the function is called.")  
        return result  
    return wrapper  
  
@my_decorator  
def greet(name):  
    print(f"Hello, {name}!")  
  
greet("Kaitlyn")
```

In this example, `my_decorator` takes a function `func` and defines a nested wrapper function. The wrapper function takes any number of positional (`*args`) and keyword arguments (`**kwargs`), calls `func` with these arguments, and performs additional actions before and after the call.

Decorators with Arguments

Sometimes, you may want your decorator to accept arguments. To achieve this, you need to add another layer of function nesting. This allows the outer function to accept arguments and return the actual decorator function. Here's an example:

```
def repeat_decorator(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat_decorator(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Kaitlyn")
```

In this example, `repeat_decorator` takes an argument `times` and returns the actual decorator function `decorator`. The `decorator` function defines the wrapper function, which repeats the call to the original function `times` times.

The `functools.wraps` Decorator

When creating decorators, it is a good practice to use `functools.wraps` to preserve the metadata (such as the name and docstring) of the original

function. Without `functools.wraps`, the metadata of the original function is lost, and the decorated function ends up with the metadata of the wrapper function. Here's how you can use `functools.wraps`:

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper called")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def example():
    """This is an example function"""
    print("Example function")

print(example.__name__) # Output: example
print(example.__doc__)  # Output: This is an example function
```

In this example, the `@wraps(func)` decorator makes sure that the wrapper function retains the name and docstring of the original `example` function.

Chaining Decorators

You can also apply multiple decorators to a single function. This is known as chaining decorators. When chaining decorators, each decorator is applied in the order they are listed. Here's an example:

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper
```

```
def exclamation_decorator(func):  
    def wrapper(*args, **kwargs):  
        result = func(*args, **kwargs)  
        return result + "!"  
    return wrapper  
  
@uppercase_decorator  
@exclamation_decorator  
def greet(name):  
    return f"Hello, {name}"  
  
print(greet("Kaitlyn")) # Output: HELLO, KAITLYN!
```

In this example, the `greet` function is first decorated with `@exclamation_decorator`, which adds an exclamation mark to the result, and then with `@uppercase_decorator`, which converts the result to uppercase.

Common Use Cases for Decorators

Decorators are used in various scenarios to improve and control the behavior of functions and methods. Here are some common use cases:

Logging

Logging is essential for debugging and monitoring applications. Decorators can be used to log function calls, parameters, and return values without cluttering the main code.

```
def log_decorator(func):  
    def wrapper(*args, **kwargs):  
        print(f"Calling {func.__name__} with arguments {args} and keyword arguments {kwargs}")  
        result = func(*args, **kwargs)  
        print(f"{func.__name__} returned {result}")  
        return result  
    return wrapper  
  
@log_decorator  
def add(a, b):
```

```
    return a + b

add(3, 5)
```

In this example, the `log_decorator` logs the function name, arguments, and return value each time the `add` function is called.

Access Control and Authentication

In web applications, access control and authentication are critical. ~~Decorators can enforce access restrictions and check user credentials before~~ allowing access to certain functions.

```
def require_authentication(func):
    def wrapper(user, *args, **kwargs):
        if not user.is_authenticated:
            raise PermissionError("User is not authenticated")
        return func(user, *args, **kwargs)
    return wrapper

@require_authentication
def get_secret_data(user):
    return "Secret Data"

class User:
    def __init__(self, authenticated):
        self.is_authenticated = authenticated

authenticated_user = User(authenticated=True)
non_authenticated_user = User(authenticated=False)

print(get_secret_data(authenticated_user))
# print(get_secret_data(non_authenticated_user)) # This would raise a PermissionError
```


In this example, the `require_authentication` decorator checks if a user is authenticated before granting access to the `get_secret_data` function.

Caching

Caching improves performance by storing the results of expensive function calls and returning the cached result when the same inputs occur again. The `functools.lru_cache` decorator provides a built-in caching mechanism.

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
print(fibonacci.cache_info())
```

In this example, the `fibonacci` function is decorated with `lru_cache`, which caches the results of the function calls, making subsequent calls with the same arguments much faster.

Timing

Timing decorators measure the time taken by a function to execute, which is useful for performance analysis and optimization.

```
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
```

```
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time} seconds to execute")
        return result
    return wrapper

@timer_decorator
def long_running_function():
    time.sleep(2)
    return "Done"

print(long_running_function())
```

In this example, the `timer_decorator` measures the execution time of the `long_running_function` and prints the duration.

Validation

Decorators can validate function arguments to make sure they meet certain criteria before the function is executed.

```
def validate_args(func):
    def wrapper(*args, **kwargs):
        for arg in args:
            if not isinstance(arg, int) or arg < 0:
                raise ValueError("All arguments must be non-negative integers")
        return func(*args, **kwargs)
    return wrapper

@validate_args
def add(a, b):
    return a + b

print(add(3, 5))
# print(add(-3, 5)) # This would raise a ValueError
```

In this example, the `validate_args` decorator checks that all arguments are non-negative integers before calling the `add` function.

Context Management

Decorators can be used to manage resources such as database connections, file handles, or network sockets, ensuring they are properly acquired and released.

```
import contextlib

@contextlib.contextmanager
def open_file(file_name, mode):
    file = open(file_name, mode)
    try:
        yield file
    finally:
        file.close()

with open_file('example.txt', 'w') as f:
    f.write('Hello, world!')
```

In this example, the `open_file` decorator manages the opening and closing of a file, ensuring the file is properly closed even if an exception occurs.

These use cases demonstrate the flexibility and power of Python decorators in improving and controlling the behavior of functions in a clean, reusable, and readable manner.

Creating Your Own Decorators

Creating custom decorators allows you to tailor functionality specifically to your needs. This section will guide you through the step-by-step process of making your own decorators, focusing on key concepts and providing

detailed examples. We will also break down some of the examples we mentioned earlier in the article to give you a clearer understanding of how they work.

Step-by-Step Guide to Creating a Decorator

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
@my_decorator  
def say_hello():  
    print("Hello!")  
  
say_hello()
```

- **Define the Decorator Function:** The first step is to define a function that will serve as your decorator. This function should take another function as its argument.

```
def my_decorator(func):  
    # ...
```

- **Create a Wrapper Function:** Inside your decorator, define a wrapper function that will add the desired behavior before and after the original function call.

```
def my_decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")
```

- **Return the Wrapper Function:** The decorator function should return the wrapper function. This is crucial because it allows the decorated function to be replaced by the wrapper function, effectively adding the new behavior.

```
def my_decorator(func):  
    def wrapper():  
        # ...  
    return wrapper
```

- **Apply the Decorator:** Use the @ symbol to apply your decorator to a function.

```
@my_decorator  
def say_hello():  
    print("Hello!")  
  
say_hello()
```

When `say_hello` is called, the `wrapper` function inside `my_decorator` is executed, showing the added behavior before and after the original function call.

In this example, we start by defining the `my_decorator` function, which takes another function `func` as an argument. Inside this decorator, we define a `wrapper` function that prints a message before and after calling the original function `func`. The decorator function then returns the `wrapper` function. By applying the decorator using the `@` syntax, the `say_hello` function is wrapped by `my_decorator`, so calling `say_hello` executes the `wrapper` function, demonstrating the added behavior.

Detailed Example with Arguments

Sometimes, you may need your decorator to accept arguments. This requires adding another layer of nesting. Let's break down the earlier example of a decorator with arguments to understand each step.

```
def repeat_decorator(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat_decorator(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Kaitlyn")
```

- **Define the Outer Function:** Start by defining an outer function that will accept the arguments for the decorator.

```
def repeat_decorator(times):  
    # ...
```

- **Create the Decorator Function:** Inside the outer function, define the decorator function that takes `func` as an argument.

```
def repeat_decorator(times):  
    def decorator(func):  
        # ...  
    return decorator
```

- **Define the Wrapper Function:** Inside the decorator function, define the wrapper function that will execute the original function multiple times.

```
def repeat_decorator(times):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            for _ in range(times):  
                result = func(*args, **kwargs)  
            return result  
        return wrapper  
    return decorator
```

- **Apply the Decorator with Arguments:** Use the `@` symbol to apply the configured decorator to a function.

```
@repeat_decorator(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Kaitlyn")
```

When `greet` is called, it prints "Hello, Kaitlyn!" three times, demonstrating how the decorator applies the repetition logic.

In this example, we created a `repeat_decorator` that takes an argument `times`, specifying how many times the decorated function should be called. The `repeat_decorator` function returns the actual decorator function, which in turn defines the `wrapper` function. This `wrapper` function uses a loop to call the original function `func` the specified number of times. By applying the decorator with `@repeat_decorator(3)`, the `greet` function is executed three times when called.

Using `functools.wraps` to Preserve Metadata

To make sure that the decorated function retains the original function's metadata (such as its name and docstring), use the `functools.wraps` decorator. Let's break down this process using an example we mentioned earlier.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper called")
        return func(*args, **kwargs)
    return wrapper
```



```
@my_decorator
def example():
    """This is an example function"""
    print("Example function")

print(example.__name__) # Output: example
print(example.__doc__)  # Output: This is an example function
```

- **Import `functools.wraps`**: Begin by importing `wraps` from the `functools` module.

```
from functools import wraps
```

- **Apply `wraps` to the Wrapper Function**: Inside your decorator, use the `@wraps` decorator on the wrapper function.

```
def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper called")
        return func(*args, **kwargs)
    return wrapper
```

- **Verify Metadata Preservation**: Apply the decorator and verify that the metadata is preserved.

```
@my_decorator
def example():
    """This is an example function"""
```

```
print("Example function")

print(example.__name__) # Output: example
print(example.__doc__)  # Output: This is an example function
```

Using `functools.wraps` make sure that `example.__name__` and `example.__doc__` contain the correct metadata from the original function.

In this example, we improve the `my_decorator` by using the `@wraps(func)` decorator from the `functools` module. The `@wraps` decorator is applied to the wrapper function inside `my_decorator`. This ensures that when `my_decorator` is used to decorate a function, the metadata of the original function (such as its name and docstring) is preserved. The example demonstrates this by printing out the `__name__` and `__doc__` attributes of the decorated `example` function, showing that they match those of the original function.

Chaining Multiple Decorators

You can apply multiple decorators to a single function. When chaining decorators, each decorator is applied in the order they are listed. Let's revisit the chaining decorators example to break down each step.

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result + "!"
    return wrapper

@uppercase_decorator
```

```
@exclamation_decorator
def greet(name):
    return f"Hello, {name}"

print(greet("Kaitlyn")) # Output: HELLO, KAITLYN!
```

- **Define Multiple Decorators:** First we create multiple decorator functions.

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result + "!"
    return wrapper
```

- **Apply Multiple Decorators:** Use the @ symbol to apply multiple decorators to a function.

```
@uppercase_decorator
@exclamation_decorator
def greet(name):
    return f"Hello, {name}"

print(greet("Kaitlyn")) # Output: HELLO, KAITLYN!
```

In this example, we have two decorators: **uppercase_decorator** and **exclamation_decorator**. The **uppercase_decorator** converts the result of the

function to uppercase, while the `exclamation_decorator` adds an exclamation mark to the result. When these decorators are chained using the `@` syntax, they are applied in the order listed. This means `exclamation_decorator` is applied first, adding the exclamation mark, and then `uppercase_decorator` is applied, converting the entire result to uppercase. The final output of the `greet` function when called with "Kaitlyn" is "HELLO, KAITLYN!".

By following these steps, you can create custom decorators to extend the functionality of your functions in a clean and reusable way. Whether you need simple behavior modifications or complex configurations, decorators provide a powerful tool to improve your Python code.

Conclusion

Python decorators are a powerful feature that allows you to extend the behavior of your functions and methods in a clean and readable way. Throughout this article, we've explored what decorators are, how they work, and how you can create your own to suit your specific needs. We've also examined common use cases such as logging, access control, caching, and more, demonstrating how decorators can improve the modularity and maintainability of your code. By understanding and utilizing decorators, you can write more efficient, readable, and flexible Python programs.

1. [Official Python Documentation on Decorators](#)
2. [Official documentation on `functools.wraps`](#)

Thank you for reading! If you find this article helpful, please consider highlighting, clapping, responding or connecting with me on Twitter/X as it's very appreciated and helps keeps content like this free!

Python

Decorators

Programming

Technology

Software Development

Some rights reserved ⓘ

Open in app ↗

Medium

 Search Write 14**Written by Alexander Obregon**

25K Followers · 15 Following

Follow



Software engineer posting daily about programming topics. Sharing what I learn along the way. Recaps and more:
<https://alexanderobregon.substack.com>

Responses (1)



José Gregorio Castillo Pacheco

What are your thoughts?

Jpilli
Apr 4

Thank you Alexander for such a comprehensive article on Python Decorators with easy to follow examples.

The common use cases that you have covered, helped me visualise the situations where I could write my own Python Decorators.

Your example on how to... [more](#)



15



1 reply

[Reply](#)

More from Alexander Obregon



Alexander Obregon

Enhancing Logging with @Log and @Slf4j in Spring Boot Applications

Introduction

Sep 21, 2023



292



5



Alexander Obregon

Java Memory Leaks: Detection and Prevention

Introduction

Nov 12, 2023



758



6



Alexander Obregon

Using Spring's @Retryable Annotation for Automatic Retries



Alexander Obregon

Navigating Client-Server Communication with Spring's...

Software systems are unpredictable, with challenges like network delays and third-...

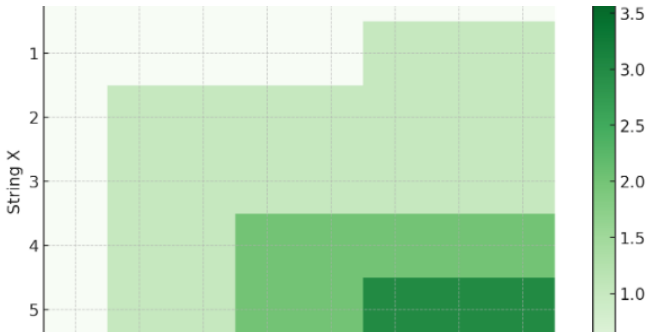
Introduction

Sep 16, 2023👤 392💬 8🔖+⋮

Sep 3, 2023👤 196💬 4🔖+⋮

See all from Alexander Obregon

Recommended from Medium



 Connie Zhou

Mastering Data Algorithms—Part 18 Dynamic Programming in...

Dynamic Programming (DP) is an optimization technique used to solve...

Nov 4, 2024

🔖+⋮

6d ago👤 4

🔖+⋮


Tool	uv Equivalent	Improvement
pip	<code>uv pip install</code>	8–10x faster
<code>/ venv</code>	<code>uv venv</code>	Instant creation
<code>uv pip compile</code>		Faster resolution and lockfile support
	<code>uv pip sync</code>	Reproducible installs
<code>requirements.txt</code>	<code>uv uses requirements.lock</code>	Modern lockfile support

 Nafiul Khan Earth

NextGen Python Development using uv

What is uv?

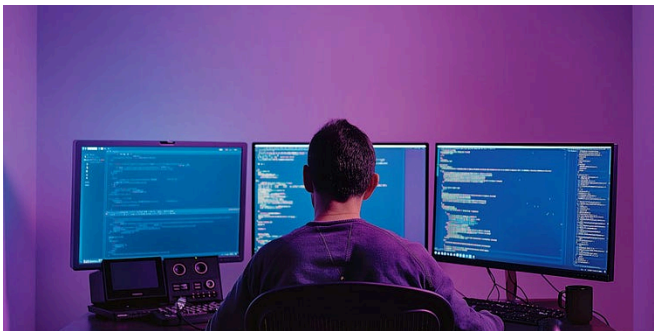



 Ethan

10 Python Tips and Tricks to Write Cleaner, Faster Code

Python’s simplicity and versatility make it a favorite among developers, but mastering it...

Apr 10



 In 딜리버스 by Hyunil Kim

Understanding Python’s asyncio: A Deep Dive into the Event Loop


Python’s asyncio is a powerful library that enables concurrent programming through...

Dec 18, 2024



2



 In Python in Plain English by Stephen Odogwu

Mimicking a List of Dictionaries with a Linked List in Python

I will be starting this article with my thought process and visualization. An experiment(yo...

Nov 7, 2024



12



 Saro

Loops in Python {1}

Hello,



Dec 12, 2024



1



See more recommendations