Python Decorators: A Comprehensive Guide

The article introduces the amazingly powerful syntactic sugar of Python: decorators.

Marcin Kozak

.

Published in

Towards Data Science

.

11 min read
.

Oct 19

Spacejoy on Unsplash

Python decorators are one of those concepts that seem simple if you understand them, but very difficult otherwise. Many Python beginners see them as a magical tool that they must learn and use in their own code in order to perform true magic. But using built-in decorators or those from site packages is not enough; it's like doing magic with a magic box bought in a kid shop. True magic comes from writing your own decorators.

I remember when I desperately wanted to learn how to write and use my own decorators in a real project, not just for fun. When that time finally came, the pleasure I felt was immense. This experience made me eager to find more opportunities to implement my own decorators.

I hope that after reading this article, you will have no problems with Python decorators. Therefore, this article aims to introduce the concept of Python decorators to those who have not been able to understand it yet. I hope to reveal the magic behind them in a simple to understand way.

There is much more to decorators than that. We will discuss the basics, but the good news is that these basics should be enough for you to implement even complex and functional decorators. In future articles, we will go further into the details by discussing the intricacies of decorators and their various use cases.

Python programmers at all levels can benefit from this article. *Beginners* will learn the basics of decorators, while *intermediates* will gain a deeper understanding; *advanced programmers* can use the article to refresh their memory. In addition, sometimes it's good to look at a particular concept from different angles, not only the one we've been using for years — here, I am offering my viewpoint on decorators and their usefulness, which I hope will be helpful to readers of all levels.

Introduction to decorators

Python decorators are a powerful and versatile tool, but they should be used judiciously to avoid overuse and misuse. A decorator is a function that allows you to modify the behavior of another function. Decorators can also be written as classes, but this is less common and will not be covered in this article.

When you have a decorator, say `my_decorator()`, you use it to decorate another function, say `foo()`, in the following way:

```
@my_decorator
def foo(x, y):
    # do something; in result,
    # you obtain changed_x and changed_y
    return changed_x, changed_y
```

After decorating the `foo()` function, we no longer know what it does without also knowing what the `my_decorator` decorator does. The decorator may add new behavior, such as logging, or it may completely change the function's behavior and return value. For example, the decorated `foo()` function may return a dictionary instead of a tuple, or it may return `None`. To find out what the `foo()` function does after being decorated, we must check the definition of the `my_decorator` decorator.

I don't know the etymology of the word "decorator" used in the Python context. In Python, the word comes from the decorator pattern, but this does not explain the original etymology. If you know it, please share it with us in the comments.

I personally find Python decorators to be a beautiful form of syntactic sugar. It is no wonder they are called decorators, as they decorate the function being decorated. I appreciate both the appearance and power of decorators.

However, I am aware of the difficulty that decorators can introduce. As explained above, we cannot know how a decorated function behaves without knowing the definition of its decorator. Additionally, the fact that multiple decorators can be used on a single function can make things even more complex.

Let me organize the idea behind decorators into three steps:

1. *Need*. You have a function, but you need to change its behavior. This can result from various reasons. For instance, you may need to add logging to all functions in an application or changing the behavior of a function from an external module.
2. *Definition*. You write a decorator function that is responsible for this updated behavior. It can take one or more parameters in addition to the original function. The decorator function typically calls the original function, but it doesn’t have to.
3. *Use*. You overwrite the original function with the new one. This can be done in two ways: via decoration or assignment, but decoration is much more common. Calling the decorated function using its original name means calling the new function, because the original function no longer exists unless it was copied.

Each of these three steps is equally important, so let’s discuss these three steps one by one.

Step 1: Need

Okay, so you *need* to change the behavior of a function.

Why use a decorator to change the behavior of a function instead of simply rewriting it? There are a few reasons:

- You may not be able to rewrite the function. For example, it may be a function from an external module.
- You may not want to rewrite the function. For example, it may be a large or complex function, or it may be a function that is used in many different places.
- It may be cumbersome to rewrite the function. For example, it may be called by many different functions.
- You may only need to change the behavior of the function for development, but you want to use the original function in production.
- You may need to change the behavior of many functions. In this case, it is much more efficient to write a decorator than to rewrite each function individually.

The most common scenario is when you need to change the behavior of many functions. In this case, you can write one decorator and apply it to all of the functions that you need to change. This can save you a lot of time and code.

Here are a few examples of how decorators can be used:

- To add logging to all existing functions in a project.
- To measure and log the execution time of each function in an app.
- To add authentication and authorization to functions called in an app.
- To cache output returned from functions.
- To change a function from writing data to a local file to writing data to a remote database.

- To silence a particular function for testing. For example, you could use a decorator to prevent a function from writing to a remote database during testing.

Decorators are also heavily used in mocking in Python. Mocking allows you to create fake objects that simulate the behavior of real objects. This can be useful for testing code that relies on external resources, such as databases and web services.

Step 2: Definition

Have you noticed that after decorating a function with the `@` syntax, the decorated function is called using the name of the original function? This is what makes a function a decorator. If the original function, say `foo()`, is still available as `foo()`, and there is a new function, say `foo_changed()`, whose behavior is the changed behavior of `foo()`, *there is no decoration involved here*. So, decorating a function involves overwriting the original function. You can keep a copy of the original function under a new name, but the original function itself has been replaced by the decorated function.

Decorating involves overwriting the original function.

It's time to leave the world of abstraction and move to practicalities. Let's create a simple decorator, called `scream()`, that makes functions scream:

```
from typing import Callable
(1) def scream(func: Callable) -> Callable:
(2)     def inner(*args, **kwargs):
(3)         print("SCREAM!!!")
(4)         return func(*args, **kwargs)
(5)     return inner
```

This decorator takes a callable (`func`) as input and returns a callable (`inner`). The inner function prints "SCREAM!!!" to the console before calling the original function (`func`). I gave up a docstring to make the code shorter, but in real work, you should definitely add docstrings to your decorators.

In real work, you should definitely add docstrings to your decorators.

Here is a breakdown of the `scream()` decorator line by line:

1. `def scream(func: Callable) -> Callable:` → This is the function signature. The `scream()` decorator can be used to decorate any callable (`func`), and it returns a callable, too.[1]
2. `def inner(*args, **kwargs):` → This is the inner function. It prints "SCREAM!!!"; you can use whatever name you want, but I usually make it `inner`, like here. When decorating a function, you can use any parameters that the decorated function takes. For example, the `scream()` decorator can be used for any function, with any number of parameters of any types (hence the use of `*args, **kwargs`).
3. `print("SCREAM!!!")` → This is the new behavior, added to the original behavior of the decorated function. Whatever the function does, it will *first* scream (by printing `"SCREAM!!!"`), and then it will do what it was originally supposed to do. Note that in this decorator, the new behavior is added *before*, but it can be added *after*, both *before and after*, or even *instead of* the original behavior.

4. `return func(*args, **kwargs)` → This is the original behavior of the function.
5. `return inner` → A standard line of any decorator: returning the `inner` function. This means that when the `scream()` decorator is used, it will replace the original function with the inner function.

In Appendix 1, you will find two other version of the `scream()` decorator:

- Screaming two times: before and after the original behavior
- Screaming instead of what the original function was supposed to do.

Step 3: Use

To decorate a function, you can use a decorator in two ways. This subsection describes both of them.

Method 1: Decorator used as `@decorator`

First, we need a function to decorate. Let's use two functions to illustrate that you can use the same decorator for as many functions as you want or need, and for very different functions at that.

The first function we want to decorate is `foo()`:

```
def foo():
    return "foo tells you that life is great"
```

When we call this function, we will see the following output:

```
>>> foo()
'foo tells you that life is great'
```

This is almost the easiest Python function: it takes no parameters and returns a string. This is the second function we want to decorate, `bar()`:

```
from typing import List
def bar(
    x: int,
    string: str,
    func: Callable = lambda a, b: a * b,
    **kwargs
) -> List[str]:
    """Applies a callable to each character in the given string.
    It does so, passing in the given integer and any additional
    keyword arguments. Returns a list of the results.
    """
    return [func(x, s_i, **kwargs) for s_i in string]
```

This function is more complex than `foo()`. It has three parameters: an integer `x`, a string `string`, and a callable `func`. It also accepts any additional keyword arguments. The function applies `func()` to each character in `string` and `x`, passing in the additional keyword arguments, if any. Finally, it returns a list of the results.

A simple call can look like that:

```
>>> bar(3, "abc")
['aaa', 'bbb', 'ccc']
```

Let's change use a different callable for `func`:

```
>>> def concatenate(i: int, s: str, sep: Optional[str] = "-") -> str:
...     return f"{str(i)}{sep}{s}"
>>> bar(5, "abc", func=concatenate)
['3-a', '3-b', '3-c']
>>> bar(3, "abc", func=concatenate, sep=":")
['3:a', '3:b', '3:c']
```

For our purposes, it doesn't matter what `foo()` and `bar()` do. What's important is that `foo()` is a very simple function, while `bar()` is more complex, despite being concise.

We can decorate both functions using the decorator syntax, which is shown in the two lines starting with the @ character. For the sake of completeness and clarity, I will show the full code, so I will repeat the functions' code:

```
from typing import Callable, List
@scream
def foo():
    return "foo tells you that life is great"
@scream
def bar(
    x: int,
    string: str,
    func: Callable = lambda a, b: a * b,
    **kwargs
) -> List[str]:
    return [func(x, s_i, **kwargs) for s_i in string]
```

Let's run both functions:

```
>>> foo()
SCREAM!!!
'foo tells you that life is great'
>>> bar(5, "abc", func=concatenate)
SCREAM!!!
['3-a', '3-b', '3-c']
>>> bar(3, "abc", func=concatenate, sep=":")
SCREAM!!!
['3:a', '3:b', '3:c']
```

Method 2: Using a decorator as a `function()`

The most common way to use a decorator function is as a decorator. However, there is another way, which is less common:

```
def foo():
    return "foo tells you that life is great"
foo = scream(foo)
```

In this way, you simply call the decorator function with the function to be decorated as an argument.

Irrespective of which method you use to apply the `scream()` decorator, when you run `foo()`, the decorated function will scream and then do what it was originally used to do.

Conclusion

This article explains the basics of Python decorators. I have tried to be comprehensive, but there are still many intricacies of decorators that we have not discussed. We will explore these in future articles.

There are several reasons why it is important to learn about Python decorators, not only how to use them but also how to write new ones. First, decorators are a powerful tool that can help you write concise and readable code. If you know how to write custom decorators, you will often find that they can save you a lot of time and effort.

Second, decorators can be used to quickly update legacy code. For example, if you need to change the behavior of a function or several functions, but revising the functions themselves is not an option, you can use a decorator instead. While it is always possible to rewrite the functions, if the changed behavior is the same for all of them, a single decorator may be sufficient.

Third, decorators are one of the most important syntactic sugar items in Python. If you do not understand them, you will likely be considered a Python beginner. I cannot imagine a Python developer who does not know how to use decorators, let alone understand them.

Finally, decorators are very common in the Python code base. If you are not familiar with decorators, you will not be able to understand much of the existing code.

Therefore, all intermediate and advanced Python developers should know the concept of decorators, understand how to use them, and be able to write them.

Although decorators may seem difficult at first glance, I believe that if you have read this article up to this point, you will agree that once you understand the basics, they are not such a big puzzle after all. In fact, they can be a rather simple and handy coding tool.

Footnotes

[1] For simplicity, I will use the term “decorator function” instead of “decorator callable.” However, please keep in mind that this is just a shorthand for decorators defined in both ways: as a function and as a class. I simply don’t want to overuse the word “callable,” even though it is frequently used in Python texts.

Appendices

Appendix 1: Two other versions of the `scream()` decorator

Version 2: Scream before and after running the function

```
from typing import Callable
def scream(func: Callable) -> Callable:
    def inner(*args, **kwargs):
        print("SCREAM!!!")
        output = func(*args, **kwargs)
        print("SCREAM AGAIN!!!")
        return output
    return inner
```

You would see the following output from `foo()` after decorating it with the above decorator:

```
>>> foo()
SCREAM!!!
'foo tells you that life is great'
SCREAM AGAIN!!!
```

Version 3: Scream instead of running the function

```
from typing import Callable
def scream(func: Callable) -> Callable:
    def inner(*args, **kwargs):
        print("SCREAM, JUST SCREAM!!!")
    return inner
```

And:

```
>>> foo()
SCREAM, JUST SCREAM!!!
```

This version of the `scream()` decorator completely overwrites the decorated function’s original behavior. The decorated function now only screams, and its original behavior is completely removed. This structure can be very useful in many different situations. For example, you could use it to totally silence a function:

```
from typing import Callable
def silence(func: Callable) -> Callable:
    def inner(*args, **kwargs):
        pass
    return inner
```

You can see an example of such a silencer in the code of the `easycheck` Python package:

easycheck/easycheck/easycheck.py at master · nyggus/easycheck

A module offering Python functions for simple and readable assertion-like checks to be used inside code, but also in…

github.com

Look in this code for a `switch` function.

You will also see there that you can stack decorators; here’s an example from the above `easycheck` library:

```
@switch
@make_it_true_assertion
def assert_paths(*args: Any, handle_with: type = AssertionError, **kwargs: Any)
-> None:
    return check_if_paths_exist(*args, handle_with=handle_with, **kwargs)
```

We will discuss such intricacies of decorators in future articles.

Thanks for reading. If you enjoyed this article, you may also enjoy other articles I wrote; you will see them here. And if you want to join Medium, please use my referral link below:

Join Medium with my referral link - Marcin Kozak

Read every story from Marcin Kozak (and thousands of other writers on Medium). Your membership fee directly supports…

medium.com