**Python Decorators: A Step-By-Step Introduction**

Python’s decorators allow you to extend and modify the behavior of a callable (functions, methods, and classes) without permanently modifying the callable itself.

Any sufficiently generic functionality you can “tack on” to an existing class or function’s behavior makes a great use case for decoration. This includes:

* logging,
* enforcing access control and authentication,
* instrumentation and timing functions,
* rate-limiting,
* caching; and more.

## Why Should I Master Decorators in Python?

That’s a fair question. After all, what I just mentioned sounded quite abstract and it might be difficult to see **how decorators can benefit you in your day-to-day work as a Python developer**. Here’s an example:

Imagine you’ve got 30 functions with business logic in your report-generating program. One rainy Monday morning your boss walks up to your desk and says:

“Happy Monday! Remember those TPS reports? I need you to add input/output logging to each step in the report generator. XYZ Corp needs it for auditing purposes. Oh, and I told them we can ship this by Wednesday.”

Depending on whether or not you’ve got a solid grasp on Python’s decorators, this request will either send your blood pressure spiking—or leave you relatively calm.

**Without decorators** you might be spending the next three days scrambling to modify each of those 30 functions and clutter them up with manual logging calls. Fun times.

**If you do know your decorators**, you’ll calmly smile at your boss and say:

“Don’t worry Jim, I’ll get it done by 2pm today.”

Right after that you’ll type the code for a generic @audit\_log decorator (that’s only about 10 lines long) and quickly paste it in front of each function definition. Then you’ll commit your code and grab another cup of coffee.

I’m dramatizing here. But only a little. Decorators can be that powerful 🙂

I’d go as far as to say that understanding decorators is a milestone for any serious Python programmer. They require a solid grasp of several advanced concepts in the language—including the properties of first-class functions.

But:

## Understanding Decorators Is Worth It 💡

The payoff for understanding how decorators work in Python is huge.

Sure, decorators are relatively complicated to wrap your head around for the first time—but they’re a highly useful feature that you’ll often encounter in third-party frameworks and the Python standard library.

Explaining decorators is also a make or break moment for any good Python tutorial. I’ll do my best here to introduce you to them step by step.

Before you dive in, now would be an excellent moment to refresh your memory on the properties of first-class functions in Python. I wrote a tutorial on them here on dbader.org and I would encourage you to take a few minutes to review it. The most important “first-class functions” takeaways for understanding decorators are:

* **Functions are objects**—they can be assigned to variables and passed to and returned from other functions; and
* **Functions can be defined inside other functions**—and a child function can capture the parent function’s local state (lexical closures.)

Alright, ready to do this? Let’s start with some:

## Python Decorator Basics

Now, what are decorators really? They “decorate” or “wrap” another function and let you execute code before and after the wrapped function runs.

Decorators allow you to define reusable building blocks that can change or extend the behavior of other functions. And they let you do that without permanently modifying the wrapped function itself. The function’s behavior changes only when it’s decorated.

Now what does the implementation of a simple decorator look like? In basic terms, a decorator is a callable that takes a callable as input and returns another callable.

The following function has that property and could be considered the simplest decorator one could possibly write:

def null\_decorator(func):

return func

As you can see, null\_decorator is a callable (it’s a function), it takes another callable as its input, and it returns the same input callable without modifying it.

Let’s use it to decorate (or wrap) another function:

def greet():

return 'Hello!'

greet = null\_decorator(greet)

>>> greet()

'Hello!'

In this example I’ve defined a greet function and then immediately decorated it by running it through the null\_decorator function. I know this doesn’t look very useful yet (I mean we specifically designed the null decorator to be useless, right?) but in a moment it’ll clarify how Python’s decorator syntax works.

Instead of explicitly calling null\_decorator on greet and then reassigning the greet variable, you can use Python’s @ syntax for decorating a function in one step:

@null\_decorator

def greet():

return 'Hello!'

>>> greet()

'Hello!'

Putting an @null\_decorator line in front of the function definition is the same as defining the function first and then running through the decorator. Using the @ syntax is just syntactic sugar, and a shortcut for this commonly used pattern.

Note that using the @ syntax decorates the function immediately at definition time. This makes it difficult to access the undecorated original without brittle hacks. Therefore you might choose to decorate some functions manually in order to retain the ability to call the undecorated function as well.

So far, so good. Let’s see how:

## Decorators Can Modify Behavior

Now that you’re a little more familiar with the decorator syntax, let’s write another decorator that actually does something and modifies the behavior of the decorated function.

Here’s a slightly more complex decorator which converts the result of the decorated function to uppercase letters:

def uppercase(func):

def wrapper():

original\_result = func()

modified\_result = original\_result.upper()

return modified\_result

return wrapper

Instead of simply returning the input function like the null decorator did, this uppercase decorator defines a new function on the fly (a closure) and uses it to wrap the input function in order to modify its behavior at call time.

The wrapper closure has access to the undecorated input function and it is free to execute additional code before and after calling the input function. (Technically, it doesn’t even need to call the input function at all.)

Note how up until now the decorated function has never been executed. Actually calling the input function at this point wouldn’t make any sense—you’ll want the decorator to be able to modify the behavior of its input function when it gets called eventually.

Time to see the uppercase decorator in action. What happens if you decorate the original greet function with it?

@uppercase

def greet():

return 'Hello!'

>>> greet()

'HELLO!'

I hope this was the result you expected. Let’s take a closer look at what just happened here. Unlike null\_decorator, our uppercase decorator returns a different function object when it decorates a function:

>>> greet

<function greet at 0x10e9f0950>

>>> null\_decorator(greet)

<function greet at 0x10e9f0950>

>>> uppercase(greet)

<function uppercase.<locals>.wrapper at 0x10da02f28>

And as you saw earlier, it needs to do that in order to modify the behavior of the decorated function when it finally gets called. The uppercase decorator is a function itself. And the only way to influence the “future behavior” of an input function it decorates is to replace (or wrap) the input function with a closure.

That’s why uppercase defines and returns another function (the closure) that can then be called at a later time, run the original input function, and modify its result.

Decorators modify the behavior of a callable through a wrapper so you don’t have to permanently modify the original. The callable isn’t permanently modified—its behavior changes only when decorated.

This let’s you “tack on” reusable building blocks, like logging and other instrumentation, to existing functions and classes. It’s what makes decorators such a powerful feature in Python that’s frequently used in the standard library and in third-party packages.

## Applying Multiple Decorators to a Single Function

Perhaps not surprisingly, you can apply more than one decorator to a function. This accumulates their effects and it’s what makes decorators so helpful as reusable building blocks.

Here’s an example. The following two decorators wrap the output string of the decorated function in HTML tags. By looking at how the tags are nested you can see which order Python uses to apply multiple decorators:

def strong(func):

def wrapper():

return '<strong>' + func() + '</strong>'

return wrapper

def emphasis(func):

def wrapper():

return '<em>' + func() + '</em>'

return wrapper

Now let’s take these two decorators and apply them to our greet function at the same time. You can use the regular @ syntax for that and just “stack” multiple decorators on top of a single function:

@strong

@emphasis

def greet():

return 'Hello!'

What output do you expect to see if you run the decorated function? Will the @emphasis decorator add its <em> tag first or does @strong have precedence? Here’s what happens when you call the decorated function:

>>> greet()

'<strong><em>Hello!</em></strong>'

This clearly shows in what order the decorators were applied: from bottom to top. First, the input function was wrapped by the @emphasis decorator, and then the resulting (decorated) function got wrapped again by the @strong decorator.

To help me remember this bottom to top order I like to call this behavior decorator stacking. You start building the stack at the bottom and then keep adding new blocks on top to work your way upwards.

If you break down the above example and avoid the @ syntax to apply the decorators, the chain of decorator function calls looks like this:

decorated\_greet = strong(emphasis(greet))

Again you can see here that the emphasis decorator is applied first and then the resulting wrapped function is wrapped again by the strong decorator.

This also means that deep levels of decorator stacking will have an effect on performance eventually because they keep adding nested function calls. Usually this won’t be a problem in practice, but it’s something to keep in mind if you’re working on performance intensive code.

## Decorating Functions That Accept Arguments

All examples so far only decorated a simple nullary greet function that didn’t take any arguments whatsoever. So the decorators you saw here up until now didn’t have to deal with forwarding arguments to the input function.

If you try to apply one of these decorators to a function that takes arguments it will not work correctly. How do you decorate a function that takes arbitrary arguments?

This is where Python’s \*args and \*\*kwargs feature for dealing with variable numbers of arguments comes in handy. The following proxy decorator takes advantage of that:

def proxy(func):

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

There are two notable things going on with this decorator:

* It uses the \* and \*\* operators in the wrapper closure definition to collect all positional and keyword arguments and stores them in variables (args and kwargs).
* The wrapper closure then forwards the collected arguments to the original input function using the \* and \*\* “argument unpacking” operators.

(It’s a bit unfortunate that the meaning of the star and double-star operators is overloaded and changes depending on the context they’re used in. But I hope you get the idea.)

Let’s expand the technique laid out by the proxy decorator into a more useful practical example. Here’s a trace decorator that logs function arguments and results during execution time:

def trace(func):

def wrapper(\*args, \*\*kwargs):

print(f'TRACE: calling {func.\_\_name\_\_}() '

f'with {args}, {kwargs}')

original\_result = func(\*args, \*\*kwargs)

print(f'TRACE: {func.\_\_name\_\_}() '

f'returned {original\_result!r}')

return original\_result

return wrapper

Decorating a function with trace and then calling it will print the arguments passed to the decorated function and its return value. This is still somewhat of a toy example—but in a pinch it makes a great debugging aid:

@trace

def say(name, line):

return f'{name}: {line}'

>>> say('Jane', 'Hello, World')

'TRACE: calling say() with ("Jane", "Hello, World"), {}'

'TRACE: say() returned "Jane: Hello, World"'

'Jane: Hello, World'

Speaking of debugging—there are some things you should keep in mind when debugging decorators:

## How to Write “Debuggable” Decorators

When you use a decorator, really what you’re doing is replacing one function with another. One downside of this process is that it “hides” some of the metadata attached to the original (undecorated) function.

For example, the original function name, its docstring, and parameter list are hidden by the wrapper closure:

def greet():

"""Return a friendly greeting."""

return 'Hello!'

decorated\_greet = uppercase(greet)

If you try to access any of that function metadata you’ll see the wrapper closure’s metadata instead:

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

>>> decorated\_greet.\_\_name\_\_

'wrapper'

>>> decorated\_greet.\_\_doc\_\_

None

This makes debugging and working with the Python interpreter awkward and challenging. Thankfully there’s a quick fix for this: the functools.wraps decorator included in Python’s standard library.

You can use functools.wraps in your own decorators to copy over the lost metadata from the undecorated function to the decorator closure. Here’s an example:

import functools

def uppercase(func):

@functools.wraps(func)

def wrapper():

return func().upper()

return wrapper

Applying functools.wraps to the wrapper closure returned by the decorator carries over the docstring and other metadata of the input function:

@uppercase

def greet():

"""Return a friendly greeting."""

return 'Hello!'

>>> greet.\_\_name\_\_

'greet'

>>> greet.\_\_doc\_\_

'Return a friendly greeting.'

As a best practice I’d recommend that you use functools.wraps in all of the decorators you write yourself. It doesn’t take much time and it will save you (and others) debugging headaches down the road.

## Python Decorators – Key Takeaways

* Decorators define reusable building blocks you can apply to a callable to modify its behavior without permanently modifying the callable itself.
* The @ syntax is just a shorthand for calling the decorator on an input function. Multiple decorators on a single function are applied bottom to top (decorator stacking).
* As a debugging best practice, use the functools.wraps helper in your own decorators to carry over metadata from the undecorated callable to the decorated one.