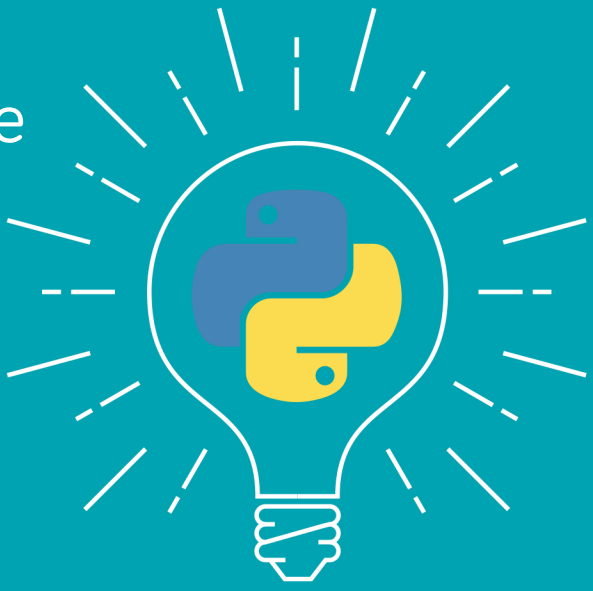
PYTHON TRICKS THE BOOK

A Buffet of Awesome Python Features



Dan Bader

Python Tricks: The Book

Dan Bader

For online information and ordering of this and other books by Dan Bader, please visit realpython.com. For more information, please contact Dan Bader at info@realpython.com.

ISBN: 9781775093305 (paperback)

ISBN: 9781775093312 (electronic)

Cover design by Anja Pircher Design (anjapircher.com)

Updated 2018-01-14 I would like to thank Michael Howitz, Johnathan Willitts, Julian Orbach, Johnny Giorgis, Bob White, Daniel Meyer, Michael Stueben, Smital Desai, Andreas Kreisig, David Perkins, Jay Prakash Singh, Ben Felder, and Steve Schmerler for their excellent feedback.

3.3 The Power of Decorators

At their core, 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 the following:

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

Now, why should you master the use of decorators in Python? 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. Let me try to bring some clarity to this question by giving you a somewhat real-world 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, right?

If you do know your decorators however, 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*.

I believe that the payoff for understanding how decorators work in Python can be enormous.

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 however, now would be an excellent moment to refresh your memory on the properties of *first-class functions* in Python. There's a chapter on them in this book, 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
- **Functions can be defined inside other functions**—and a child function can capture the parent function's local state (lexical closures)

Alright, are you ready to do this? Let's get started.

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*.

What might 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 you 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 this example will clarify how Python's special-case 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 more conveniently:

```
@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.

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 eventually gets called.

You might want to let that sink in for a minute or two. I know how complicated this stuff can seem, but we'll get it sorted out together, I promise.

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 0x76da02f28>
```

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 closure so you don't have to permanently modify the original. The original callable isn't permanently modified—its behavior changes only when decorated.

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

A Quick Intermission

By the way, if you feel like you need a quick coffee break or a walk around the block at this point—that's totally normal. In my opinion

closures and decorators are some of the most difficult concepts to understand in Python.

Please, take your time and don't worry about figuring this out immediately. Playing through the code examples in an interpreter session one by one often helps make things sink in.

I know you can do it!

Applying Multiple Decorators to a 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 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 eventually have an effect on performance because they keep adding nested function calls. In practice, this usually won't be a problem, but it's something to keep in mind if you're working on performance-intensive code that frequently uses decoration.

Decorating Functions That Accept Arguments

All examples so far only decorated a simple *nullary* `greet` function that didn't take any arguments whatsoever. Up until now, the decorators you saw here 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[3] 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.

[3] cf. "Fun With *args and **kwargs" chapter

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.[4]

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:

[4] cf. Python Docs: "functools.wraps"

```
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.

Oh, and congratulations—you've made it all the way to the end of this complicated chapter and learned a whole lot about decorators in Python. Great job!

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.
- Just like any other tool in the software development toolbox, decorators are not a cure-all and they should not be overused. It's important to balance the need to "get stuff done" with the goal of "not getting tangled up in a horrible, unmaintainable mess of a code base."