Share this

LEARN PYTHON PROGRAMMING

How to write decorators in Python

Jose Salvatierra
8 min read · Aug 29

Decorators in Python can be a confusing topic, particularly for newer students. That's why we've made a four-part video course for you to go through, linked below.

Decorators in Python • 18 min

I've also put together some written explanations of the important concepts, so if you're more of a reader, or if you just want more examples, you'll find all of the information you need below.

Higher order functions

Before learning about decorators, you must know a little bit about higher order functions. After all, all decorators are higher order functions (but not all higher order functions are decorators!).

So what is are higher order functions?

They're just functions that take other functions as arguments. Higher order functions usually call the functions passed to them, but they generally involve additional operations as well.

For example, let's say you have a function called `say_hello`:

```python
def say_hello():
    user_name = input("Enter your name: ")
    print(f"Hello, {user_name}!")
```

Let's say that you want to create another function that runs this one, but also prints a welcome message before running.

You could do this:

Share this

```
    print("Welcome to my application!")
    say_hello()
```

This is not a higher order function. It's just a function!

If you wanted to be able to print the welcome message before any arbitrary function, you should create a higher order function that takes in another function as an argument. Like so:

```
def welcome_user(func):
    print("Welcome to my application!")
    func()
```

The benefit of this is that now any function can be passed to `welcome_user`, so it is very reusable. This particular example may not seem terribly useful, but we'll look at a more interesting one later on in this post!

You'd use this like so:

```
def welcome_user(func):
    print("Welcome to my application!")
    func()

def say_hello():
    user_name = input("Enter your name: ")
    print(f"Hello, {user_name}!")

welcome_user(say_hello)  # Calls both functions and prints both things.
```

What are decorators?

A decorator is a higher order function that doesn't directly call the argument; instead, it creates another function that does, and returns that.

That's a lot of "function" in one phrase, so let's break it down!

Here's what a decorator might look like:

```
def decorator():
    def say_goodbye():
        print("Goodbye!")

    return say_goodbye
```

The `decorator` function defines an inner function, `say_goodbye`, and then returns it. The inner function doesn't run until it is called. You could use it like this:

```
my_variable = decorator()
my_variable()  # Goodbye!
```

Although it seems that `my_variable` is not a function (no `def` keyword there!), `my_variable` is very much a function.

Remember that the right side of an assignment is evaluated before the

Share this

`decorator`, the `say_goodbye` function, is assigned to `my_variable`.

So, is `decorator` a decorator?

Not yet.

In order to be a decorator, it needs to take a function as an argument:

```
def decorator(func):
    def say_goodbye():
        func()
        print("Goodbye!")

    return say_goodbye
```

You could then use this decorator like so:

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

greet = decorator(say_hello)
greet()  # Hello! and Goodbye!
```

Now `decorator` is worthy of its name. It takes a function as an argument and returns a new function that essentially "extends" it. Instead of just calling the original function, the new `greet` function does *more*.

Usually you'd use decorators to replace existing functions and simultaneously extend them:

```python
def say_hello():
    print("Hello!")

say_hello = decorator(say_hello)
say_hello()  # Hello! and Goodbye!
```

In that last example, the `decorator` function runs first since it's the right-hand side of the assignment. The original `say_hello` function will be called only within the `say_goodbye` function defined inside the decorator.

After line 4, the `say_hello` variable in the global scope no longer refers to the original function. Instead it refers to the `say_goodbye` function.

You can check this by printing out the `__name__`:

```python
def say_hello():
    print("Hello!")

say_hello = decorator(say_hello)
print(say_hello.__name__)  # say_goodbye
```

This is a drawback of these incomplete decorators, in that the old function's `__name__` and `__doc__` properties are replaced by those of the new function. At the end of the day, we're completely replacing the variable with a new value, so this makes sense.

However often when using decorators we want to extend the `say_hello` function but let it keep its name and documentation (if it has any). This is where the built-in module `functools` comes in.

documentation:

```python
import functools

def decorator(func):
    @functools.wraps(func)
    def say_goodbye():
        func()
        print("Goodbye!")

    return say_goodbye
```

This now means the name is preserved:

```python
def say_hello():
    print("Hello!")

say_hello = decorator(say_hello)
print(say_hello.__name__)  # say_hello
```

But, wait! What's that `@` syntax?

Well, let's learn about it!

The `@` syntax for decorators

The `@` syntax is used to apply a decorator. When we wrote `@functools.wraps(func)` earlier, we applied the `wraps` decorator to our

Share this

We can use that syntax with our code too. At the moment, our code looks like this:

```
import functools


def decorator(func):
    @functools.wraps(func)
    def say_goodbye():
        func()
        print("Goodbye!")

    return say_goodbye


def say_hello():
    print("Hello!")


say_hello = decorator(say_hello)
say_hello()
```

But using the `@` syntax, it can be simplified to this:

```
import functools


def decorator(func):
    @functools.wraps(func)
    def say_goodbye():
        func()
        print("Goodbye!")

    return say_goodbye
```

Share this

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


say_hello()
```

Note that we've gotten rid of the line that re-assigned `say_hello`, and now instead we've placed `@decorator` above the function definition.

This does exactly the same: it creates the function, and then passes it to the decorator, and re-assigns it.

If you run the code, you'll see that calling `say_hello()` still prints out both "Hello!" and "Goodbye!".

A key benefit of this is that the function is immediately decorated, so it cannot be called accidentally before it has been decorated.

Decorating functions with parameters

At this point, our `say_hello` function is being replaced with `say_goodbye`. That means that if `say_hello` had any parameters, those are lost when we replace the function. See this example:

```
import functools


def decorator(func):
    @functools.wraps(func)
    def say_goodbye():
```

Share this

```python
    return say_goodbye


@decorator
def say_hello(name):
    print(f"Hello, {name}!")


say_hello("Bob")  # TypeError, say_goodbye() takes 0 positional arguments but 1 was
```

Indeed if we were to do this, we would need to add the parameter to `say_goodbye` as well:

```python
def decorator(func):
    @functools.wraps(func)
    def say_goodbye(name):
        func(name)
        print("Goodbye!")
    return say_goodbye
```

But this is a bad idea, because it couples our decorator to our `say_hello` function, and that means we cannot use that decorator with other functions that have different numbers of parameters.

Instead the recommended approach is to generalise our decorators by making them take any number of arguments, and pass arguments (if any) to the decorated function as well.

We'll use the standard syntax `*args, **kwargs` in order to do that. If you haven't encountered them before, here's a quick guide:

```
def decorator(func):
    @functools.wraps(func)
    def say_goodbye(*args, **kwargs):
        func(*args, **kwargs)
        print("Goodbye!")
    return say_goodbye
```

Now this decorator can be used with any function, no matter how many parameters they have!

Writing decorators with parameters

In order to learn about decorators with parameters, let's take a look at another example. This example is also covered in the videos linked at the top of this post, so do check those out if you haven't already!

Let's say you have a function, `get_admin_password`, that returns the password to the admin panel, and another function, `get_user_password`, that returns the password to the user dashboard. They looks like this:

```
def get_admin_password():
    return "1234"

def get_user_password():
    return "secure_password"
```

You can call one of these functions like so:

But naturally, you'd like only administrators to be able to get the admin password, and users to be able to get the user panel password.

This is what your user looks like:

```
user = {"name": "Bob Smith", "access_level": "admin"}
```

So, let's write a decorator that secures the `get_password` function so it's not exposed to any user!

We'll start by only allowing administrators to run the function:

```
def make_secure(func):
    def secure_func(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)

    return secure_func
```

A simple decorator, but uses everything we've talked about until now!

It defines an inner function, which will run the passed function, `func` , only if the user's access level is "admin" at the time.

Here's the complete code:

Share this

```
import functools

user = {"name": "Bob Smith", "access_level": "admin"}


def make_secure(func):
    @functools.wraps(func)
    def secure_func(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)

    return secure_func


@make_secure
def get_admin_password():
    return "1234"

@make_secure
def get_user_password():
    return "secure_password"

print(get_password("admin"))  # 1234
print(get_password("user"))  # None
```

Why does it print `None` when we try to get the user panel password? Because the decorator never runs `return func(*args, **kwargs)`, so it just returns `None` by default—as all Python functions do.

At the moment, the `make_secure` decorator only allows users to run the function if their access level is `"admin"`. It does not allow users to run either function.

Indeed, we need two slightly different decorators, one that checks for an `"admin"` level of permissions, and one that checks for a `"user"` level of

But there would be a *lot* of duplication if we just define two decorators, so instead what we'll do is make them more dynamic by adding a parameter:

```python
@make_secure("admin")
def get_admin_password():
    return "1234"

@make_secure("user")
def get_user_password():
    return "secure_password"
```

If we are to do this, we need to make some changes to the decorator itself!

Something *very* important about this new bit of syntax is that `make_secure("admin")` will run first, and then it will be applied as a decorator with the `@` syntax.

So our decorator must change into this:

```python
def make_secure(access_level):
    def decorator(func):
        @functools.wraps(func)
        def secure_func(*args, **kwargs):
            if user["access_level"] == "admin":
                return func(*args, **kwargs)

        return secure_func
    return decorator
```

Share this

We have added one more level, because `make_secure` is no longer a decorator *per se*. It is now a function that creates a decorator, and then *that* is applied to the functions with the 'at' syntax.

So when you run `@make_secure("admin")`, that'll first of all execute the `make_secure` function and create the `decorator` function.

`@decorator` is then applied, but it already knows about the access level you provided.

This is all quite confusing, and it's one of the most convoluted pieces of code you'll see in Python. Very seldom do we have three-level deep nesting of functions one inside of another!

I'd recommend checking the video playlist for another explanation, with slightly different examples. The more you see this kind of structure, the more sense it makes, and the easier it is to understand!

Wrapping Up

If you're interested in improving your Python skills, and learning about many more Python topics, we'd love to have you on our Complete Python Course. It's a fully comprehensive course, over 30 hours long, covering all major Python topics! We think it's an awesome course, but if it's not for you, we also have a 30 day money back guarantee.

You can also sign up to our mailing list below, as we send discount codes to our subscribers every month!

Hope to see you there!

Share this

Jose Salvatierra

I'm a software engineer turned instructor! I founded Teclado to help me do this for everyone. I hope you enjoy the content!

Read More

Learn Python Programming

The @property decorator in Python →

How to simplify long if statements in Python →

Basics of Python a Beginner Should Know - Part 1 →

→ See all 149 posts

Share this

PYTHON SNIPPETS

Formatting Numbers for Printing in Python

Learn how to leverage Python's detailed formatting language to control how numbers are displayed in your applications.

Phil Best
3 min read · Sep 2

Like what you see? Enter your e-mail to hear when new posts come out!

E-mail*

Name*

Receive our newsletter and get notified about new posts we publish on the site, as well as occasional special offers.

Sign Up