TNW | NEWS

The heart of tech

C++

Decorators in Python will make your code so much better

Analyze, test, and re-use your code with little more than an @ symbol

NEWS

STORY BY
Rhea Moutafis

If there's one thing that makes Python incredibly successful, that would be its readability. Everything else hinges on that: if code is unreadable, it's hard to maintain. It's also not beginner-friendly then — a novice getting boggled by unreadable code won't attempt writing its own one day.

Python was already readable and beginner-friendly before decorators came around. But as the language started getting used for more and more things, Python developers felt the need for more and more features, without cluttering the landscape and making code unreadable.

Decorators are a prime-time example of a perfectly implemented feature. It does take a while to wrap your head around, but it's worth it. As you start using them, you'll notice how they don't overcomplicate things and make your code neat and snazzy.

Before anything else: higher-order functions

In a nutshell, decorators are a neat way to handle higher-order functions. So let's look at those first!

Functions returning functions

bine the two? Think about it a minute before you look below.

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


def simon(func):
    return func("Simon")


simon(greet)
```

The output is 'Hello, Simon!'. Hope that makes sense to ya!

Of course, we could have just called greet("Simon"). However, the whole point is that we might want to put “Simon” into many different functions. And if we don’t use “Simon” but something more complicated, we can save a whole lot of lines of code by packing it into a function like simon().

Functions inside other functions

We can also define functions inside other functions. That’s important because decorators will do that, too! Without decorators it looks like this:

```
def respect(maybe):
    def congrats():
        return "Congrats, bro!"
    def insult():
        return "You're silly!"


    if maybe == "yes":
        return congrats
    else:
        return insult
```

The function respect() returns a function; respect("yes") returns the congrats function, respect("brother") (or some other argument instead of "brother") returns the insult function. To call the functions, enter respect("yes")() and respect("brother")(), just like a normal function.

Got it? Then you’re all set for decorators!

Let’s try a combination of the two previous concepts: a function that takes another function and defines a function. Sounds mind-boggling? Consider this:

```
def startstop(func):
    def wrapper():
        print("Starting...")
        func()
        print("Finished!")
    return wrapper


def roll():
    print("Rolling on the floor laughing XD")


roll = startstop(roll)
```

The last line ensures that we don’t need to call startstop(roll)() anymore; roll() will suffice. Do you know what the output of that call is? Try it yourself if you’re unsure!

Now, as a very good alternative, we could insert this right after defining startstop():

```
@startstop
def roll():
    print("Rolling on the floor laughing XD")
```

This does the same, but glues roll() to startstop() at the onset.

Added flexibility

Why is that useful? Doesn’t that consume exactly as many lines of code as before?

In this case, yes. But once you’re dealing with slightly more complicated stuff, it gets really useful. For once, you can move all decorators (i.e. the def startstop() part above) into its own module. That is, you write them into a file called decorators.py and write something like this into your main file:

```
@startstop
def roll():
    print("Rolling on the floor laughing XD")
```

In principle, you can do that without using decorators. But this way it makes life easier because you don't have to deal with nested functions and endless bracket-counting anymore.

You can also nest decorators:

```
from decorators import startstop, exectime


@exectime
@startstop
def roll():
    print("Rolling on the floor laughing XD")
```

Note that we haven't defined exectime() yet, but you'll see it in the next section. It's a function that can measure how long a process takes in Python.

This nesting would be equivalent to a line like this:

```
roll = exectime(startstop(roll))
```

Bracket counting is starting! Imagine you had five or six of those functions nested inside each other. Wouldn't the decorator notation be much easier to read than this nested mess?

You can even use decorators on functions that accept arguments. Now imagine a few arguments in the line above and your chaos would be complete. Decorators make it neat and tidy.

Finally, you can even add arguments to your decorators — like @mydecorator(argument). Yeah, you can do all of this without decorators. But then I wish you a lot of fun understanding your decorator-free code when you re-read it in three weeks…

Applications: where decorators cut the cream

Now that I've hopefully convinced you that decorators make your life three times easier, let's look at some classic examples where decorators are basically indispensable.

just use a decorator!

```
import time


def measuretime(func):
    def wrapper():
        starttime = time.perf_counter()
        func()
        endtime = time.perf_counter()
        print(f"Time needed: {endtime - starttime} seconds")
    return wrapper


@measuretime
def wastetime():
    sum([i**2 for i in range(1000000)])


wastetime()
```

A dozen lines of code and we're done! Plus, you can use measuretime() on as many functions as you want.

Slowing code down

Sometimes you don't want to execute code immediately but wait a while. That's where a slow-down decorator comes in handy:

```
import time


def sleep(func):
    def wrapper():
        time.sleep(300)
        return func()
    return wrapper


@sleep
def wakeup():
    print("Get up! Your break is over.")


wakeup()
```

Testing and debugging

Say you have a whole lot of different functions that you call at different stages, and you're losing the overview over what's being called when. With a simple decorator for every function definition, you can bring more clarity. Like so:

```
def debug(func):
    def wrapper():
        print(f"Calling {func.__name__}")
    return wrapper


@debug
def scare():
    print("Boo!")


scare()
```

There is a more elaborate example here. Note, though, that to understand that example, you'll have to check how to decorate functions with arguments. Still, it's worth the read!

Reusing code

This kinda goes without saying. If you've defined a function decorator(), you can just sprinkle @decorator everywhere in your code. To be honest, I don't think it gets any simpler than that!

Handling logins

If you have functionalities that should only be accessed if a user is logged in, that's also fairly easy with decorators. I'll refer you to the full example for reference, but the principle is quite simple: first, you define a function like login_required(). Before any function definition that needs logging in, you pop @login_required. Simple enough, I'd say.

Syntactic sugar — or why Python is so sweet

It's not like I'm not critical of Python or not using alternative languages where it's appropriate. But there's a big allure to Python: it's so easy to digest, even when you're not a computer scientist by training and just want to make things work.

NEWS

But I hope you've come to see why it's such a big sweet-factor. Syntactic sugar to add some pleasure to your life! Without health risks, except for having your eyes glued on a screen.

This article was written by Rhea Moutafis and was originally published on Towards Data Science. You can read it here.

Also tagged with

PYTHON PYTHON DEVELOPERS PYTHON DECORATORS NESTING

Published July 10, 2021 - 2:00 pm UTC

Back to top

POPULAR ON TNW TODAY

1 Hands-on: LeMond's carbon fiber ebikes are lightweight works of art

2 This EV drove a record-breaking 710km on a single charge

3 I'm loving the new Rocksmith beta, but definitely not uninstalling RS2014 yet

4 Do you really need Google Drive's new desktop app?

5 These Google search tips will make finding stuff online way easier

NEWS

SHAREABLES

Try this brain-melting 3D game that was built entirely in ASCII

HARDFORK

C++ creat
made

The heart of tech

NEWS