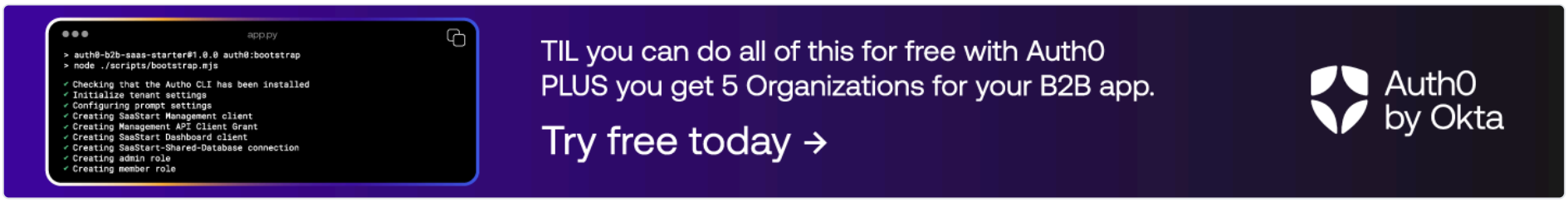
Primer on Python Decorators

by Geir Arne Hjelle Dec 14, 2024 233 Comments intermediate python

Mark as Completed Share

Table of Contents



Remove ads

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Python Decorators 101**

Python decorators allow you to modify or extend the behavior of functions and methods without changing their actual code. When you use a Python decorator, you wrap a function with another function, which takes the original function as an argument and returns its modified version. This technique provides a simple way to implement higher-order functions in Python, enhancing code reusability and readability.

You can use decorators in various practical scenarios, such as logging, enforcing access control, caching results, or measuring execution time. To create a custom decorator, define a function that takes another function as an argument, creates a nested wrapper function, and returns the wrapper. When applying a decorator, you place `@decorator` on the line before the function definition.

By the end of this tutorial, you'll understand that:

- **Python decorators** allow you to wrap a function with another function to extend or modify its behavior without altering the original function's code.
- **Practical use cases** for decorators include logging, enforcing access control, caching results, and measuring execution time.
- **Custom decorators** are written by defining a function that takes another function as an argument, defines a nested wrapper function, and returns the wrapper.
- **Multiple decorators** can be applied to a single function by stacking them, one per line, before the function definition.
- The **order of decorators** impacts the final output, as each decorator wraps the next, influencing the behavior of the decorated function.

You can find all the examples from this tutorial by downloading the accompanying materials below:

Get Your Code: Click here to download the free sample code that shows you how to create and use Python decorators.

Free Bonus: Click here to get access to a free "The Power of Python Decorators" guide that shows you three advanced decorator patterns and techniques you can use to write cleaner and more Pythonic programs.

Decorators Cheat Sheet: Click here to get access to a free three-page Python decorators cheat sheet that summarizes the techniques explained in this tutorial.

Decorators Q&A Transcript: Click here to get access to a 25-page chat log from our Python decorators Q&A session in the Real Python Community Slack where we discussed common decorator questions.

Take the Quiz: Test your knowledge with our interactive “Decorators” quiz. You’ll receive a score upon completion to help you track your learning progress:

Interactive Quiz

Decorators

In this quiz, you'll revisit the foundational concepts of what Python decorators are and how to create and use them.

Python Functions

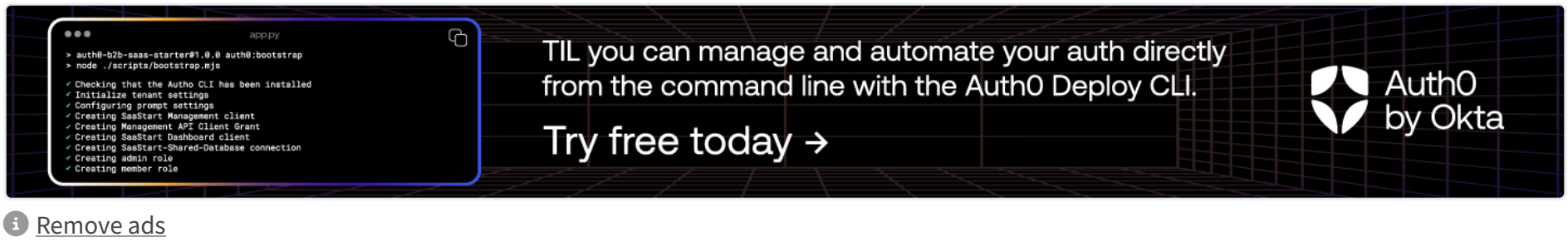
In order to understand decorators, you must first understand some finer points of how functions work. There are many aspects to functions, but in the context of decorators, **a function returns a value based on the given arguments**. Here’s a basic example:

Python

```
>>> def add_one(number):
...     return number + 1
...

>>> add_one(2)
3
```

In general, functions in Python may also have side effects rather than just turning an input into an output. The `print()` function is an example of this: it returns `None` while having the side effect of outputting something to the console. However, to understand decorators, it’s enough to think about functions as tools that turn given arguments into values.



Remove ads

First-Class Objects

In functional programming, you work almost entirely with pure functions that don’t have side effects. While not a purely functional language, Python supports many functional programming concepts, including treating functions as first-class objects.

This means that *functions can be passed around and used as arguments*, just like any other object like `str`, `int`, `float`, `list`, and so on. Consider the following three functions:

Python greeters.py

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we're the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Bob")
```

Here, `say_hello()` and `be_awesome()` are regular functions that expect a name given as a string. The `greet_bob()` function, however, expects a function as its argument. You can, for example, pass it the `say_hello()` or the `be_awesome()` function.

To test your functions, you can run your code in interactive mode. You do this with the `-i` flag. For example, if your code is in a file named `greeters.py`, then you run `python -i greeters.py`:

Python

```
>>> greet_bob(say_hello)
'Hello Bob'

>>> greet_bob(be_awesome)
'Yo Bob, together we're the awesomest!'
```

Note that `greet_bob(say_hello)` refers to two functions, `greet_bob()` and `say_hello`, but in different ways. The `say_hello` function is named without parentheses. This means that only a reference to the function is passed. The function isn't executed. The `greet_bob()` function, on the other hand, is written with parentheses, so it will be called as usual.

This is an important distinction that's crucial for how functions work as first-class objects. A function name without parentheses is a reference to a function, while a function name with trailing parentheses calls the function and refers to its return value.

Inner Functions

It's possible to define functions *inside other functions*. Such functions are called inner functions. Here's an example of a function with two inner functions:

Python `inner_functions.py`

```
def parent():
    print("Printing from parent()")

    def first_child():
        print("Printing from first_child()")

    def second_child():
        print("Printing from second_child()")

    second_child()
    first_child()
```

What happens when you call the `parent()` function? Think about this for a minute. Then run `inner_functions.py` in interactive mode to try it out. The output will be as follows:

Python

```
>>> parent()
Printing from parent()
Printing from second_child()
Printing from first_child()
```

Note that the order in which the inner functions are defined does not matter. Like with any other functions, the printing only happens when the inner functions are executed.

Furthermore, the inner functions aren't defined until the parent function is called. They're locally scoped to `parent()`, meaning they only exist inside the `parent()` function as local variables. Try calling `first_child()`. You'll get an error:

Python

```
>>> first_child()
Traceback (most recent call last):
  ...
NameError: name 'first_child' is not defined
```

Whenever you call `parent()`, the inner functions `first_child()` and `second_child()` are also called. But because of their local scope, they aren't available outside of the `parent()` function.

Functions as Return Values

Python also allows you to return functions from functions. In the following example, you rewrite `parent()` to return one of the inner functions:

Python inner_functions.py

```
def parent(num):
    def first_child():
        return "Hi, I'm Elias"

    def second_child():
        return "Call me Ester"

    if num == 1:
        return first_child
    else:
        return second_child
```

Note that you're returning `first_child` without the parentheses. Recall that this means that you're *returning a reference to the function* `first_child`. In contrast, `first_child()` with parentheses refers to the result of evaluating the function. You can see this in the following example:

Python

```
>>> first = parent(1)
>>> second = parent(2)

>>> first
<function parent.<locals>.first_child at 0x7f599f1e2e18>

>>> second
<function parent.<locals>.second_child at 0x7f599dad5268>
```

The somewhat cryptic output means that the `first` variable refers to the local `first_child()` function inside of `parent()`, while `second` points to `second_child()`.

You can now use `first` and `second` as if they're regular functions, even though you can't directly access the functions they point to:

Python

```
>>> first()
'Hi, I'm Elias'

>>> second()
'Call me Ester'
```

You recognize the return values of the inner functions that you defined inside of `parent()`.

Finally, note that in the earlier example, you executed the inner functions within the parent function—for example, `first_child()`. However, in this last example, you didn't add parentheses to the inner functions, such as `first_child`, upon returning. That way, you got a reference to each function that you could call in the future.

Master Real-World Python Skills
With a Community of Experts

Level Up With Unlimited Access to Our Vast Library of Python Tutorials and Video Lessons

Watch Now »

Remove ads

Simple Decorators in Python

Now that you've seen that functions are just like any other object in Python, you're ready to move on and see the magical beast that is the Python decorator. You'll start with an example:

Python hello_decorator.py

```python
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = decorator(say_whee)
```

Here, you've defined two regular functions, `decorator()` and `say_whee()`, and one inner `wrapper()` function. Then you redefined `say_whee()` to apply `decorator()` to the original `say_whee()`.

Can you guess what happens when you call `say_whee()`? Try it in a REPL. Instead of running the file with the `-i` flag, you can also import the function manually:

Python

```
>>> from hello_decorator import say_whee

>>> say_whee()
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

To understand what's going on here, look back at the earlier examples. You're applying everything that you've learned so far.

The so-called decoration happens at the following line:

Python

```
say_whee = decorator(say_whee)
```

In effect, the name `say_whee` now points to the `wrapper()` inner function. Remember that you return `wrapper` as a function when you call `decorator(say_whee)`:

Python

```
>>> say_whee
<function decorator.<locals>.wrapper at 0x7f3c5dfd42f0>
```

However, `wrapper()` has a reference to the original `say_whee()` as `func`, and it calls that function between the two calls to `print()`.

Put simply, *a decorator wraps a function, modifying its behavior.*

Before moving on, have a look at a second example. Because `wrapper()` is a regular Python function, the way a decorator modifies a function can change dynamically. So as not to disturb your neighbors, the following example will only run the decorated code during the day:

Python quiet_night.py

```python
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
```

If you try to call `say_whee()` after bedtime, nothing will happen:

Python

```
>>> from quiet_night import say_whee

>>> say_whee()
```

Here, `say_whee()` doesn't print any output. That's because the `if` test failed, so the wrapper didn't call `func()`, the original `say_whee()`.

Adding Syntactic Sugar

Look back at the code that you wrote in `hello_decorator.py`. The way you decorated `say_whee()` is a little clunky. First of all, you end up typing the name `say_whee` three times. Additionally, the decoration gets hidden away below the definition of the function.

Instead, Python allows you to *use decorators in a simpler way with the `@` symbol*, sometimes called the pie syntax. The following example does the exact same thing as the first decorator example:

Python `hello_decorator.py`

```python
def decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@decorator
def say_whee():
    print("Whee!")
```

So, `@decorator` is just a shorter way of saying `say_whee = decorator(say_whee)`. It's how you apply a decorator to a function.

Remove ads

Reusing Decorators

Recall that a decorator is just a regular Python function. All the usual tools for reusability are available. Now, you'll create a module where you store your decorators and that you can use in many other functions.

Create a file called `decorators.py` with the following content:

Python `decorators.py`

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

The `do_twice()` decorator calls the decorated function twice. You'll soon see the effect of this in several examples.

> **Note:** You can name your inner function whatever you want, and a generic name like `wrapper()` is usually okay. You'll see a lot of decorators in this tutorial. To keep them apart, you'll name the inner function with the same name as the decorator but with a `wrapper_` prefix.

You can now use this new decorator in other files by doing a regular import:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def say_whee():
...     print("Whee!")
...
```

When you run this example, you should see that the original `say_whee()` is executed twice:

Python

```
>>> say_whee()
Whee!
Whee!
```

There are two *Whee!* exclamations printed, confirming that `@do_twice` does what it says on the tin.

Free Bonus: Click here to get access to a free "The Power of Python Decorators" guide that shows you three advanced decorator patterns and techniques you can use to write cleaner and more Pythonic programs.

Decorating Functions With Arguments

Say that you have a function that accepts some arguments. Can you still decorate it? Give it a try:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def greet(name):
...     print(f"Hello {name}")
...
```

You now apply `@do_twice` to `greet()`, which expects a name. Unfortunately, calling this function raises an error:

Python

```
>>> greet(name="World")
Traceback (most recent call last):
  ...
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

The problem is that the inner function `wrapper_do_twice()` doesn't take any arguments, but you passed `name="World"` to it. You could fix this by letting `wrapper_do_twice()` accept one argument, but then it wouldn't work for the `say_whee()` function that you created earlier.

The solution is to use *args and **kwargs in the inner wrapper function. Then it'll accept an arbitrary number of positional and keyword arguments. Rewrite `decorators.py` as follows:

Python `decorators.py`

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

The `wrapper_do_twice()` inner function now accepts any number of arguments and passes them on to the function that it decorates. Now both your `say_whee()` and `greet()` examples work. Start a fresh REPL:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def say_whee():
...     print("Whee!")
...

>>> say_whee()
Whee!
Whee!

>>> @do_twice
... def greet(name):
...     print(f"Hello {name}")
...

>>> greet("World")
Hello World
Hello World
```

You use the same decorator, `@do_twice`, to decorate two different functions. This hints at one of the powers of decorators. They add behavior that can apply to many different functions.

Remove ads

Returning Values From Decorated Functions

What happens to the return value of decorated functions? Well, that's up to the decorator to decide. Say you decorate a simple function as follows:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def return_greeting(name):
...     print("Creating greeting")
...     return f"Hi {name}"
...
```

Try to use it:

Python

```
>>> hi_adam = return_greeting("Adam")
Creating greeting
Creating greeting

>>> print(hi_adam)
None
```

Oops, your decorator ate the return value from the function.

Because the `do_twice_wrapper()` doesn't explicitly return a value, the call `return_greeting("Adam")` ends up returning `None`.

To fix this, you need to *make sure the wrapper function returns the return value of the decorated function*. Change your `decorators.py` file:

Python `decorators.py`

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

Now you return the return value of the last call of the decorated function. Check out the example again:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def return_greeting(name):
...     print("Creating greeting")
...     return f"Hi {name}"
...

>>> return_greeting("Adam")
Creating greeting
Creating greeting
'Hi Adam'
```

This time, `return_greeting()` returns the greeting `'Hi Adam'`.

Finding Yourself

A great convenience when working with Python, especially in the interactive shell, is its powerful introspection ability.

Introspection is the ability of an object to know about its own attributes at runtime. For instance, a function knows its own name and documentation:

Python

```
>>> print
<built-in function print>

>>> print.__name__
'print'

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    <full help message>
```

When you inspect `print()`, you can see its name and documentation. The introspection works for functions that you define yourself as well:

Python

```
>>> say_whee
<function do_twice.<locals>.wrapper_do_twice at 0x7f43700e52f0>

>>> say_whee.__name__
'wrapper_do_twice'

>>> help(say_whee)
Help on function wrapper_do_twice in module decorators:

wrapper_do_twice()
```

However, after being decorated, `say_whee()` has gotten very confused about its identity. It now reports being the `wrapper_do_twice()` inner function inside the `do_twice()` decorator. Although technically true, this isn't very useful information.

To fix this, decorators should use the `@functools.wraps` decorator, which will preserve information about the original function. Update `decorators.py` again:

Python `decorators.py`

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
```

You don't need to change anything about the decorated `say_whee()` function, but you need to restart your REPL to see the effect:

Python

```
>>> from decorators import do_twice

>>> @do_twice
... def say_whee():
...     print("Whee!")
...

>>> say_whee
<function say_whee at 0x7ff79a60f2f0>

>>> say_whee.__name__
'say_whee'

>>> help(say_whee)
Help on function say_whee in module whee:

say_whee()
```

Much better! Now `say_whee()` is still itself after decoration.

> **Note:** The `@functools.wraps` decorator uses `functools.update_wrapper()` to update special attributes like `__name__` and `__doc__` that are used in the introspection.

You've now learned the basics of how to create a decorator. However, `@do_twice` isn't a very exciting decorator, and there aren't a lot of use cases for it. In the next section, you'll implement several decorators that illustrate what you know so far and that you can use in your own code.

Remove ads

A Few Real World Examples

You'll now look at a few more useful examples of decorators. You'll notice that they'll mainly follow the same pattern that you've learned so far:

Python

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

This formula is a good boilerplate template for building more complex decorators.

You'll continue to store your decorators in `decorators.py`. Recall that you can download all the examples in this tutorial:

Get Your Code: Click here to download the free sample code that shows you how to create and use Python decorators.

Timing Functions

You'll start by creating a `@timer` decorator. It'll measure the time a function takes to execute and then print the duration to the console. Here's the code:

Python decorators.py

```
1  import functools
2  import time
3
4  # ...
5
6  def timer(func):
7      """Print the runtime of the decorated function"""
8      @functools.wraps(func)
9      def wrapper_timer(*args, **kwargs):
10         start_time = time.perf_counter()
11         value = func(*args, **kwargs)
12         end_time = time.perf_counter()
13         run_time = end_time - start_time
14         print(f"Finished {func.__name__}() in {run_time:.4f} secs")
15         return value
16     return wrapper_timer
```

This decorator works by storing the time just before the function starts running in line 10 and just after the function finishes in line 12. The runtime of the function is then the difference between the two, calculated in line 13. You use `time.perf_counter()`, which does a good job of measuring time intervals.

Now, add `waste_some_time()` as an example of a function that spends some time, so that you can test `@timer`. Here are some examples of timings:

Python

```
>>> from decorators import timer

>>> @timer
... def waste_some_time(num_times):
...     for _ in range(num_times):
...         sum([number**2 for number in range(10_000)])
...

>>> waste_some_time(1)
Finished waste_some_time() in 0.0010 secs

>>> waste_some_time(999)
Finished waste_some_time() in 0.3260 secs
```

Run it yourself. Work through the definition of `@timer` line by line. Make sure you understand how it works. Don't worry if you don't get everything, though. Decorators are advanced beings. Try to sleep on it or make a drawing of the program flow.

> **Note:** The `@timer` decorator is great if you just want to get an idea about the runtime of your functions. If you want to do more precise measurements of code, then you should instead consider the `timeit` module in the standard library. It temporarily disables garbage collection and runs multiple trials to strip out noise from short function calls.

If you're interested in learning more about timing functions, then have a look at Python Timer Functions: Three Ways to Monitor Your Code.

Debugging Code

The following `@debug` decorator will print a function's arguments and its return value every time you call the function:

Python decorators.py

```python
import functools

# ...

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k}={repr(v)}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kwargs_repr)
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__}() returned {repr(value)}")
        return value
    return wrapper_debug
```

The signature is created by joining the string representations of all the argument:

- **Line 9:** You create a list of the positional arguments. Use `repr()` to get a nice string representing each argument.
- **Line 10:** You create a list of the keyword arguments. The f-string formats each argument as `key=value`, and again, you use `repr()` to represent the value.
- **Line 11:** You join together the lists of positional and keyword arguments to one signature string with each argument separated by a comma.
- **Line 14:** You print the return value after the function is executed.

It's time to see how the decorator works in practice by applying it to a simple function with one positional and one keyword argument:

Python

```
>>> from decorators import debug

>>> @debug
... def make_greeting(name, age=None):
...     if age is None:
...         return f"Howdy {name}!"
...     else:
...         return f"Whoa {name}! {age} already, you're growing up!"
...
```

Note how the `@debug` decorator prints the signature and return value of the `make_greeting()` function:

Python

```
>>> make_greeting("Benjamin")
Calling make_greeting('Benjamin')
make_greeting() returned 'Howdy Benjamin!'
'Howdy Benjamin!'

>>> make_greeting("Juan", age=114)
Calling make_greeting('Juan', age=114)
make_greeting() returned 'Whoa Juan! 114 already, you're growing up!'
'Whoa Juan! 114 already, you're growing up!'

>>> make_greeting(name="Maria", age=116)
Calling make_greeting(name='Maria', age=116)
make_greeting() returned 'Whoa Maria! 116 already, you're growing up!'
'Whoa Maria! 116 already, you're growing up!'
```

This example might not seem immediately useful since the `@debug` decorator just repeats what you wrote. It's more powerful when applied to small convenience functions that you don't call directly yourself.

The following example calculates an approximation of the [mathematical constant *e*]:

Python `calculate_e.py`

```
1  import math
2  from decorators import debug
3
4  math.factorial = debug(math.factorial)
5
6  def approximate_e(terms=18):
7      return sum(1 / math.factorial(n) for n in range(terms))
```

Here, you also apply a decorator to a function that has already been defined. In line 4, you decorate `factorial()` from the `math` standard library. You can't use the pie syntax, but you can still manually apply the decorator. The approximation of *e* is based on the following [series expansion]:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \cdots$$

When calling the `approximate_e()` function, you can see the `@debug` decorator at work:

Python

```
>>> from calculate_e import approximate_e

>>> approximate_e(terms=5)
Calling factorial(0)
factorial() returned 1
Calling factorial(1)
factorial() returned 1
Calling factorial(2)
factorial() returned 2
Calling factorial(3)
factorial() returned 6
Calling factorial(4)
factorial() returned 24
2.708333333333333
```

In this example, you get a decent approximation of the true value $e \approx 2.718281828$, adding only five terms.

Remove ads

Slowing Down Code

In this section, you'll create a decorator that slows down your code. This might not seem very useful. Why would you want to slow down your Python code?

Probably the most common use case is that you want to rate-limit a function that continuously checks whether a resource—like a web page—has changed. The `@slow_down` decorator will sleep one second before it calls the decorated function:

Python decorators.py

```
import functools
import time

# ...

def slow_down(func):
    """Sleep 1 second before calling the function"""
    @functools.wraps(func)
    def wrapper_slow_down(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_slow_down
```

In `@slow_down`, you call `time.sleep()` to have your code take a pause before calling the decorated function. To see how the `@slow_down` decorator works, you create a `countdown()` function. To see the effect of slowing down the code, you should run the example yourself:

Python

```
>>> from decorators import slow_down

>>> @slow_down
... def countdown(from_number):
...     if from_number < 1:
...         print("Liftoff!")
...     else:
...         print(from_number)
...         countdown(from_number - 1)
...

>>> countdown(3)
3
2
1
Liftoff!
```

In `countdown()`, you check if `from_number` is smaller than one. In that case, you print *Liftoff!*. If not, then you print the number and keep counting.

> **Note:** The `countdown()` function is a recursive function. In other words, it's a function calling itself. To learn more about recursive functions in Python, see Thinking Recursively in Python.

The `@slow_down` decorator always sleeps for one second. Later, you'll see how to control the rate by passing an argument to the decorator.

Registering Plugins

Decorators don't have to wrap the function that they're decorating. They can also simply register that a function exists and return it unwrapped. You can use this, for example, to create a lightweight plugin architecture:

Python `decorators.py`

```
# ...

PLUGINS = dict()

def register(func):
    """Register a function as a plug-in"""
    PLUGINS[func.__name__] = func
    return func
```

The `@register` decorator only stores a reference to the decorated function in the global `PLUGINS` dictionary. Note that you don't have to write an inner function or use `@functools.wraps` in this example because you're returning the original function unmodified.

You can now register functions as follows:

Python

```
>>> from decorators import register, PLUGINS

>>> @register
... def say_hello(name):
...     return f"Hello {name}"
...

>>> @register
... def be_awesome(name):
...     return f"Yo {name}, together we're the awesomest!"
...
```

Note that the `PLUGINS` dictionary already contains references to each function object that's registered as a plugin:

Python

```
>>> PLUGINS
{'say_hello': <function say_hello at 0x7f768eae6730>,
 'be_awesome': <function be_awesome at 0x7f768eae67b8>}
```

Python applies decorators when you define a function, so `say_hello()` and `be_awesome()` are immediately registered. You can then use `PLUGINS` to call these functions:

Python

```
>>> import random

>>> def randomly_greet(name):
...     greeter, greeter_func = random.choice(list(PLUGINS.items()))
...     print(f"Using {greeter!r}")
...     return greeter_func(name)
...

>>> randomly_greet("Alice")
Using 'say_hello'
'Hello Alice'
```

The `randomly_greet()` function randomly chooses one of the registered functions to use. In the f-string, you use the `!r` flag. This has the same effect as calling `repr(greeter)`.

The main benefit of this simple plugin architecture is that you don't need to maintain a list of which plugins exist. That list is created when the plugins register themselves. This makes it trivial to add a new plugin: just define the function and decorate it with `@register`.

If you're familiar with `globals()` in Python, then you might see some similarities to how the plugin architecture works. With `globals()`, you get access to all global variables in the current scope, including your plugins:

Python

```
>>> globals()
{..., # Many variables that aren't not shown here.
 'say_hello': <function say_hello at 0x7f768eae6730>,
 'be_awesome': <function be_awesome at 0x7f768eae67b8>,
 'randomly_greet': <function randomly_greet at 0x7f768eae6840>}
```

Using the `@register` decorator, you can create your own curated list of interesting names, effectively hand-picking some functions from `globals()`.

Remove ads

Authenticating Users

The final example before moving on to some fancier decorators is commonly used when working with a web framework. In this example, you'll use Flask to set up a `/secret` web page that should only be visible to users that are logged in or otherwise authenticated:

Python `secret_app.py`

```python
import functools
from flask import Flask, g, request, redirect, url_for

app = Flask(__name__)

def login_required(func):
    """Make sure user is logged in before proceeding"""
    @functools.wraps(func)
    def wrapper_login_required(*args, **kwargs):
        if g.user is None:
            return redirect(url_for("login", next=request.url))
        return func(*args, **kwargs)
    return wrapper_login_required

@app.route("/secret")
@login_required
def secret():
    ...
```

While this gives an idea about how to add authentication to your web framework, you should usually not write these types of decorators yourself. For Flask, you can use the Flask-Login extension instead, which adds more security and functionality.

Fancy Decorators

So far, you've seen how to create simple decorators. You already have a pretty good understanding of what decorators are and how they work. Feel free to take a break from this tutorial to practice everything that you've learned.

In the second part of this tutorial, you'll explore more advanced features, including how to do the following:

- Add **decorators to classes**
- Add **several decorators** to one function
- Create decorators with **arguments**
- Create decorators that can **optionally** take arguments
- Define **stateful** decorators
- **Define classes** that act as decorators

Ready to dive in? Here you go!

Decorating Classes

There are two different ways that you can use decorators on classes. The first one is very close to what you've already done with functions: you can *decorate the methods of a class*. This was one of the motivations for introducing decorators back in the day.

Some commonly used decorators are even built-ins in Python, including `@classmethod`, `@staticmethod`, and `@property`. The `@classmethod` and `@staticmethod` decorators are used to define methods inside a class namespace that aren't connected to a particular instance of that class. The `@property` decorator is used to customize getters and setters for class attributes. Expand the box below for an example using these decorators:

Example using built-in class decorators Show/Hide

Next, define a class where you decorate some of its methods using the `@debug` and `@timer` decorators from earlier:

Python `class_decorators.py`

```python
from decorators import debug, timer

class TimeWaster:
    @debug
    def __init__(self, max_num):
        self.max_num = max_num

    @timer
    def waste_time(self, num_times):
        for _ in range(num_times):
            sum([number**2 for number in range(self.max_num)])
```

Using this class, you can see the effect of the decorators:

Python

```
>>> from class_decorators import TimeWaster

>>> tw = TimeWaster(1000)
Calling __init__(<time_waster.TimeWaster object at 0x7efccce03908>, 1000)
__init__() returned None

>>> tw.waste_time(999)
Finished waste_time() in 0.3376 secs
```

When you create a new instance of `TimeWaster`, Python calls `.__init__()` under the hood, as your use of `@debug` reveals. The `@timer` decorator helps you monitor how much time is spent on `.waste_time()`.

The other way to use decorators on classes is to *decorate the whole class*. This is, for example, done in the `dataclasses` module:

Python

```
>>> from dataclasses import dataclass

>>> @dataclass
... class PlayingCard:
...     rank: str
...     suit: str
...
```

The meaning of the syntax is similar to the function decorators. In the example above, you could've decorated the class by writing `PlayingCard = dataclass(PlayingCard)`.

A common use of class decorators is to be a simpler alternative to some use cases of metaclasses. In both cases, you're changing the definition of a class dynamically.

Writing a class decorator is very similar to writing a function decorator. The only difference is that the decorator will receive a class and not a function as an argument. In fact, all the decorators that you saw above will work as class decorators. When you're using them on a class instead of a function, their effect might not be what you want. In the following example, the `@timer` decorator is applied to a class:

Python class_decorators.py

```python
from decorators import timer

@timer
class TimeWaster:
    def __init__(self, max_num):
        self.max_num = max_num

    def waste_time(self, num_times):
        for _ in range(num_times):
            sum([i**2 for i in range(self.max_num)])
```

Decorating a class doesn't decorate its methods. Recall that `@timer` is just shorthand for `TimeWaster = timer(TimeWaster)`. Here, `@timer` only measures the time that it takes to instantiate the class:

Python

```
>>> from class_decorators import TimeWaster

>>> tw = TimeWaster(1000)
Finished TimeWaster() in 0.0000 secs

>>> tw.waste_time(999)
```

The output from `@timer` is only shown as `tw` is created. The call to `.waste_time()` isn't timed.

Later, you'll see an example defining a proper class decorator, namely `@singleton`, which ensures that there's only one instance of a class.

Remove ads

Nesting Decorators

You can *apply several decorators* to a function at once by stacking them on top of each other:

Python

```
>>> from decorators import debug, do_twice

>>> @debug
... @do_twice
... def greet(name):
...     print(f"Hello {name}")
...
```

Think about this as the decorators being executed in the order they're listed. In other words, `@debug` calls `@do_twice`, which calls `greet()`, or `debug(do_twice(greet()))`:

Python

```
>>> greet("Yadi")
Calling greet('Yadi')
Hello Yadi
Hello Yadi
greet() returned None
```

The greeting is printed twice because of `@do_twice`. However, the output from `@debug` is only shown once, since it's called before the `@do_twice` decorator. Observe the difference if you change the order of `@debug` and `@do_twice`:

Python

```
>>> from decorators import debug, do_twice

>>> @do_twice
... @debug
... def greet(name):
...     print(f"Hello {name}")
...

>>> greet("Yadi")
Calling greet('Yadi')
Hello Yadi
greet() returned None
Calling greet('Yadi')
Hello Yadi
greet() returned None
```

Here, `@do_twice` is applied to `@debug` as well. You can see that both calls to `greet()` are annotated with debugging information.

Defining Decorators With Arguments

Sometimes, it's useful to *pass arguments to your decorators*. For instance, `@do_twice` could be extended to a `@repeat(num_times)` decorator. The number of times to execute the decorated function could then be given as an argument.

If you define `@repeat`, you could do something like this:

Python

```
>>> from decorators import repeat

>>> @repeat(num_times=4)
... def greet(name):
...     print(f"Hello {name}")
...

>>> greet("World")
Hello World
Hello World
Hello World
Hello World
```

Think about how you'd implement `@repeat`.

So far, the name written after the `@` has referred to a function object that can be called with another function. To be consistent, you then need `repeat(num_times=4)` to return a function object that can act as a decorator. Luckily, you already know how to return functions! In general, you want something like the following:

Python

```
def repeat(num_times):
    def decorator_repeat(func):
        ...  # Create and return a wrapper function
    return decorator_repeat
```

Typically, the decorator creates and returns an inner wrapper function, so writing the example out in full will give you an inner function within an inner function. While this might sound like the programming equivalent of the *Inception*, you'll untangle it all in a moment:

Python decorators.py

```
import functools

# ...

def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat
```

It looks a little messy, but you've only put the same decorator pattern that you've seen many times by now inside one additional `def` that handles the arguments to the decorator. First, consider the innermost function:

Python

```
def wrapper_repeat(*args, **kwargs):
    for _ in range(num_times):
        value = func(*args, **kwargs)
    return value
```

This `wrapper_repeat()` function takes arbitrary arguments and returns the value of the decorated function, `func()`. This wrapper function also contains the loop that calls the decorated function `num_times` times. This is no different from the earlier wrapper functions that you've seen, except that it's using the `num_times` parameter that must be supplied from the outside.

One step out, you'll find the decorator function:

Python

```
def decorator_repeat(func):
    @functools.wraps(func)
    def wrapper_repeat(*args, **kwargs):
        ...
    return wrapper_repeat
```

Again, `decorator_repeat()` looks exactly like the decorator functions that you've written earlier, except that it's named differently. That's because you reserve the base name—`repeat()`—for the outermost function, which is the one the user will call.

As you've already seen, the outermost function returns a reference to the decorator function:

Python

```
def repeat(num_times):
    def decorator_repeat(func):
        ...
    return decorator_repeat
```

There are a few subtle things happening in the `repeat()` function:

- Defining `decorator_repeat()` as an inner function means that `repeat()` will refer to a function object, `decorator_repeat`. Earlier, you used decorators like `@do_twice` without parentheses. Now, you need to add parentheses when setting up the decorator, as in `@repeat()`. This is necessary in order to add arguments.
- The `num_times` argument is seemingly not used in `repeat()` itself. But by passing `num_times`, a closure is created where the value of `num_times` is stored until `wrapper_repeat()` uses it later.

With everything set up, test your code to see if the results are as expected:

Python

```
>>> from decorators import repeat

>>> @repeat(num_times=4)
... def greet(name):
...     print(f"Hello {name}")
...

>>> greet("World")
Hello World
Hello World
Hello World
Hello World
```

That's just the result that you were aiming for.

Remove ads

Creating Decorators With Optional Arguments

With a little bit of care, you can also define *decorators that can be used both with and without arguments*. Most likely, you don't need this, but it is nice to have the flexibility. Like *Winnie-the-Pooh* says:

> Both—but don't bother about the bread, please. (Source)

As you saw in the previous section, when a decorator uses arguments, you need to add an extra outer function. The challenge now is for your code to figure out if you've called the decorator with or without arguments.

Since the function to decorate is only passed in directly if the decorator is called without arguments, the function must be an optional argument. This means that the decorator arguments must all be specified by keyword. You can enforce this with the special asterisk (`*`) syntax, which means that all the following parameters are keyword-only:

Python

```
def name(_func=None, *, key1=value1, key2=value2, ...):
    def decorator_name(func):
        ...  # Create and return a wrapper function.

    if _func is None:
        return decorator_name
    else:
        return decorator_name(_func)
```

Here, the `_func` argument acts as a marker, noting whether the decorator has been called with arguments or not:

- **Line 1:** If you've called `@name` without arguments, then the decorated function will be passed in as `_func`. If you've called it with arguments, then `_func` will be `None`, and some of the keyword arguments may have been changed from their default values. The asterisk in the argument list means that you can't call the remaining arguments as positional arguments.
- **Line 6:** In this case, you called the decorator with arguments. Return a decorator function that takes a function as an argument and returns a wrapper function.
- **Line 8:** In this case, you called the decorator without arguments. Apply the decorator to the function immediately.

Using this boilerplate on the `@repeat` decorator in the previous section, you can write the following:

Python decorators.py

```
import functools

# ...

def repeat(_func=None, *, num_times=2):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat

    if _func is None:
        return decorator_repeat
    else:
        return decorator_repeat(_func)
```

Compare this with the original `@repeat`. The only changes are the added `_func` parameter and the `if…else` block at the end.

Recipe 9.6 of the excellent *Python Cookbook* shows an alternative solution using `functools.partial()`.

You can now apply `@repeat` to different functions to test that you can now use it with or without arguments:

Python

```
>>> from decorators import repeat

>>> @repeat
... def say_whee():
...     print("Whee!")
...

>>> @repeat(num_times=3)
... def greet(name):
...     print(f"Hello {name}")
...
```

Recall that the default value of `num_times` is `2`, so using `@repeat` without any arguments is equivalent to using `@do_twice`:

Python

```
>>> say_whee()
Whee!
Whee!

>>> greet("Penny")
Hello Penny
Hello Penny
Hello Penny
```

Here, *Whee!* is repeated twice since that's the default behavior of `@repeat`. As specified by the argument, the greeting is repeated three times.

Tracking State in Decorators

Sometimes, it's useful to have *a decorator that can keep track of state*. As an example, you'll create a decorator that counts the number of times a function is called.

> **Note:** In the beginning of this guide, you learned about pure functions returning a value based on given arguments. Stateful decorators are quite the opposite, where the return value will depend on the current state, as well as the given arguments.

In the next section, you'll see how to use classes to keep state. But in simple cases, you can also get away with using function attributes:

Python `decorators.py`

```
import functools

# ...

def count_calls(func):
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__}()")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls
```

The state—the number of calls to the function—is stored in the function attribute `.num_calls` on the wrapper function. Here's the effect of using it:

Python

```
>>> from decorators import count_calls

>>> @count_calls
... def say_whee():
...     print("Whee!")
...

>>> say_whee()
Call 1 of say_whee()
Whee!

>>> say_whee()
Call 2 of say_whee()
Whee!

>>> say_whee.num_calls
2
```

You apply `@count_calls` to your old friend, `say_whee()`. Each time you call the function, you see that the call count increases. You can also manually query the `.num_calls` attribute.

Remove ads

Using Classes as Decorators

The typical way to maintain state in Python is by using classes. In this section, you'll see how to rewrite the `@count_calls` example from the previous section to *use a class as a decorator*.

Recall that the decorator syntax `@decorator` is just a quicker way of saying `func = decorator(func)`. Therefore, if `decorator` is a class, it needs to take `func` as an argument in its `.__init__()` initializer. Furthermore, the class instance needs to be callable so that it can stand in for the decorated function.

> **Note:** Up until now, all the decorators that you've seen have been defined as functions. This is how you most often will create decorators. However, you can use any callable expression as a decorator.

For a class instance to be callable, you implement the special `.__call__()` method:

Python

```
>>> class Counter:
...     def __init__(self, start=0):
...         self.count = start
...     def __call__(self):
...         self.count += 1
...         print(f"Current count is {self.count}")
...
```

The `.__call__()` method is executed each time you try to call an instance of the class:

Python

```
>>> counter = Counter()
>>> counter()
Current count is 1

>>> counter()
Current count is 2

>>> counter.count
2
```

Each time you call `counter()`, the state changes as the count increases. Therefore, a typical implementation of a decorator class should implement `.__init__()` and `.__call__()`:

Python `decorators.py`

```
import functools

# ...

class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__}()")
        return self.func(*args, **kwargs)
```

The `.__init__()` method must store a reference to the function, and it can do any other necessary initialization. The `.__call__()` method will be called instead of the decorated function. It does essentially the same thing as the `wrapper()` function in your earlier examples. Note that you need to use the `functools.update_wrapper()` function instead of `@functools.wraps`.

This `@CountCalls` decorator works the same as the one in the previous section:

Python

```
>>> from decorators import CountCalls

>>> @CountCalls
... def say_whee():
...     print("Whee!")
...

>>> say_whee()
Call 1 of say_whee()
Whee!

>>> say_whee()
Call 2 of say_whee()
Whee!

>>> say_whee.num_calls
2
```

Each call to `say_whee()` is counted and noted. In the next section, you'll look at more examples of decorators.

More Real-World Examples

You've come a long way now, having figured out how to create all kinds of decorators. You'll wrap it up, putting your newfound knowledge to use by creating a few more examples that might be useful in the real world.

Slowing Down Code, Revisited

As noted earlier, your previous implementation of `@slow_down` always sleeps for one second. Now you know how to add parameters to decorators, so you can rewrite `@slow_down` using an optional `rate` argument that controls how long it sleeps:

Python decorators.py

```
import functools
import time

# ...

def slow_down(_func=None, *, rate=1):
    """Sleep given amount of seconds before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down

    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
```

You're using the boilerplate introduced in the Creating Decorators With Optional Arguments section to make `@slow_down` callable both with and without arguments. The same recursive `countdown()` function as earlier now sleeps two seconds between each count:

Python

```
>>> from decorators import slow_down

>>> @slow_down(rate=2)
... def countdown(from_number):
...     if from_number < 1:
...         print("Liftoff!")
...     else:
...         print(from_number)
...         countdown(from_number - 1)
...
```

As before, you must run the example yourself to see the effect of the decorator:

Python

```
>>> countdown(3)
3
2
1
Liftoff!
```

There'll be a two second pause between each number in the countdown.

Creating Singletons

A singleton is a class with only one instance. There are several singletons in Python that you use frequently, including `None`, `True`, and `False`. The fact that `None` is a singleton allows you to compare for `None` using the `is` keyword, like you did when creating decorators with optional arguments:

Python

```
if _func is None:
    return decorator_name
else:
    return decorator_name(_func)
```

Using `is` returns `True` only for objects that are the exact same instance. The following `@singleton` decorator turns a class into a singleton by storing the first instance of the class as an attribute. Later attempts at creating an instance simply return the stored instance:

Python decorators.py

```
import functools

# ...

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if wrapper_singleton.instance is None:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton
```

As you see, this class decorator follows the same template as your function decorators. The only difference is that you're using `cls` instead of `func` as the parameter name to indicate that it's meant to be a class decorator.

Check it out in practice:

Python

```
>>> from decorators import singleton

>>> @singleton
... class TheOne:
...     pass
...

>>> first_one = TheOne()
>>> another_one = TheOne()

>>> id(first_one)
140094218762310

>>> id(another_one)
140094218762310

>>> first_one is another_one
True
```

By comparing object IDs and checking with the `is` keyword, you confirm that `first_one` is indeed the exact same instance as `another_one`.

> **Note:** Singleton classes aren't really used as often in Python as in other languages. The effect of a singleton is usually better implemented as a global variable inside a module.

Class decorators are less common than function decorators. You should document these well, so that your users know how to apply them.

Caching Return Values

Decorators can provide a nice mechanism for caching and memoization. As an example, look at a recursive definition of the Fibonacci sequence:

Python

```
>>> from decorators import count_calls

>>> @count_calls
... def fibonacci(num):
...     if num < 2:
...         return num
...     return fibonacci(num - 1) + fibonacci(num - 2)
...
```

While this implementation is straightforward, its runtime performance is terrible:

Python

```
>>> fibonacci(10)
<Lots of output from count_calls>
55

>>> fibonacci.num_calls
177
```

To calculate the tenth Fibonacci number, you should only need to calculate the preceding Fibonacci numbers, but this implementation somehow needs a whopping 177 calculations. It gets worse quickly: 21,891 calculations are needed for `fibonacci(20)` and almost 2.7 million calculations for the thirtieth number. This is because the code keeps recalculating Fibonacci numbers that are already known.

The usual solution is to implement Fibonacci numbers using a for loop and a lookup table. However, caching the calculations will also do the trick. First add a `@cache` decorator to your module:

Python decorators.py

```
import functools

# ...

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = {}
    return wrapper_cache
```

The cache works as a lookup table, as it stores calculations in a dictionary. You can add it to `fibonacci()`:

Python

```
>>> from decorators import cache, count_calls

>>> @cache
... @count_calls
... def fibonacci(num):
...     if num < 2:
...         return num
...     return fibonacci(num - 1) + fibonacci(num - 2)
...
```

You still use `@count_calls` to monitor the performance of your calculations. With the cache, `fibonacci()` only does the necessary calculations once:

Python

```
>>> fibonacci(10)
Call 1 of fibonacci()
...
Call 11 of fibonacci()
55

>>> fibonacci(8)
21
```

Note that in the call to `fibonacci(8)`, no new calculations were needed since the eighth Fibonacci number had already been calculated for `fibonacci(10)`.

In the standard library, a Least Recently Used (LRU) cache is available as `@functools.lru_cache`. Additionally, you can use a regular cache with `@functools.cache`.

These decorators have more features than the one you saw above. You should use `@functools.lru_cache` or `@functools.cache` instead of writing your own cache decorator.

In the next example, you don't return the result immediately. Instead, you add a call to `print()` to see when a result is calculated and not just retrieved from the cache:

Python

```
>>> import functools

>>> @functools.lru_cache(maxsize=4)
... def fibonacci(num):
...     if num < 2:
...         value = num
...     else:
...         value = fibonacci(num - 1) + fibonacci(num - 2)
...     print(f"Calculated fibonacci({num}) = {value}")
...     return value
...
```

The `maxsize` parameter specifies how many recent calls are cached. The default value is 128, but you can specify `maxsize=None` to cache all function calls. Using `@functools.cache` has the same effect as `maxsize=None`. However, be aware that this can cause memory problems if you're caching many large objects.

You can use the `.cache_info()` method to see how the cache performs, and you can tune it if needed. In your example, you used an artificially small `maxsize` to see the effect of elements being removed from the cache:

Python

```
>>> fibonacci(10)
Calculated fibonacci(1) = 1
Calculated fibonacci(0) = 0
Calculated fibonacci(2) = 1
Calculated fibonacci(3) = 2
Calculated fibonacci(4) = 3
Calculated fibonacci(5) = 5
Calculated fibonacci(6) = 8
Calculated fibonacci(7) = 13
Calculated fibonacci(8) = 21
Calculated fibonacci(9) = 34
Calculated fibonacci(10) = 55
55

>>> fibonacci(8)
21

>>> fibonacci(5)
Calculated fibonacci(1) = 1
Calculated fibonacci(0) = 0
Calculated fibonacci(2) = 1
Calculated fibonacci(3) = 2
Calculated fibonacci(4) = 3
Calculated fibonacci(5) = 5
5

>>> fibonacci(8)
Calculated fibonacci(6) = 8
Calculated fibonacci(7) = 13
Calculated fibonacci(8) = 21
21

>>> fibonacci(5)
5

>>> fibonacci.cache_info()
CacheInfo(hits=17, misses=20, maxsize=4, currsize=4)
```

In these examples, you calculate a few Fibonacci numbers. Your cache only holds four calculations at a time. For example, after calculating `fibonacci(10)`, it holds the seventh, eight, ninth, and tenth number.

Therefore, you're able to find `fibonacci(8)` without doing any recalculations. Then you ask for `fibonacci(5)`, but that fifth number has been deleted from the cache. It therefore needs to be calculated from scratch.

In most applications, you don't need to constrain your cache and can use `@functools.cache` directly.

Adding Information About Units

The following example is somewhat similar to the registering plugins example from earlier, in that it doesn't really change the behavior of the decorated function. Instead, it simply adds `unit` as a function attribute:

Python `decorators.py`

```python
# ...

def set_unit(unit):
    """Register a unit on a function"""
    def decorator_set_unit(func):
        func.unit = unit
        return func
    return decorator_set_unit
```

The following example calculates the volume of a cylinder based on its radius and height in centimeters:

Python

```
>>> import math
>>> from decorators import set_unit

>>> @set_unit("cm^3")
... def volume(radius, height):
...     return math.pi * radius**2 * height
...
```

You've added information to `volume()` that the result should be interpreted as cubic centimeters. You can later access the `.unit` function attribute when needed:

Python

```
>>> volume(3, 5)
141.3716694115407

>>> volume.unit
'cm^3'
```

Note that you could've achieved something similar using function annotations:

Python

```
>>> import math

>>> def volume(radius, height) -> "cm^3":
...     return math.pi * radius**2 * height
...
```

However, since annotations are used for type hints, it's a bit clunky to combine such units as annotations with static type checking.

Units become even more powerful and fun when connected with a library that can convert between units. One such library is `pint`. With `pint` installed (`python -m pip install Pint`), you can convert the volume to cubic inches or gallons, for example:

Python

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> vol = volume(3, 5) * ureg(volume.unit)

>>> vol
<Quantity(141.3716694115407, 'centimeter ** 3')>

>>> vol.to("cubic inches")
<Quantity(8.627028576414954, 'inch ** 3')>

>>> vol.to("gallons").m  # Magnitude
0.0373464440537444
```

You use `pint` to create a quantity that has both a magnitude and a unit. By calling `.to()`, you convert to other units. For example, the example cylinder is about 141 cubic centimeters, which translates to approximately 8.63 cubic inches and 0.0373 gallons.

You could also modify the decorator to return a `pint` `Quantity` directly. Such a `Quantity` is made by multiplying a value with the unit. In `pint`, units must be looked up in a `UnitRegistry`. You can store the registry as a function attribute on the decorator to avoid cluttering the namespace:

Python decorators.py

```python
import functools
import pint

# ...

def use_unit(unit):
    """Have a function return a Quantity with given unit"""
    use_unit.ureg = pint.UnitRegistry()
    def decorator_use_unit(func):
        @functools.wraps(func)
        def wrapper_use_unit(*args, **kwargs):
            value = func(*args, **kwargs)
            return value * use_unit.ureg(unit)
        return wrapper_use_unit
    return decorator_use_unit
```

With the `@use_unit` decorator, converting units is practically effortless:

Python

```
>>> from decorators import use_unit

>>> @use_unit("meters per second")
... def average_speed(distance, duration):
...     return distance / duration
...

>>> bolt = average_speed(100, 9.58)
>>> bolt
<Quantity(10.438413361169102, 'meter / second')>

>>> bolt.to("km per hour")
<Quantity(37.578288100208766, 'kilometer / hour')>

>>> bolt.to("mph").m  # Magnitude
23.350065679064745
```

When Usain Bolt ran 100 meters in 9.58 seconds at the 2009 world championships, he had an average speed of 10.4 meters per second. This translates to about 37.6 kilometers per hour and 23.4 miles per hour.

Validating JSON

You'll now look at one last use case. Take a quick look at the following Flask route handler:

Python

```python
@app.route("/grade", methods=["POST"])
def update_grade():
    json_data = request.get_json()
    if "student_id" not in json_data:
        abort(400)
    # Update database
    return "success!"
```

Here you ensure that the key `student_id` is part of the request. Although this validation works, it doesn't really belong in the function itself. Additionally, there may be other routes that use the same validation. So, to keep it DRY, you can abstract out any unnecessary logic with a decorator. The following `@validate_json` decorator will do the job:

Python `decorator_flask.py`

```python
import functools
from flask import abort

def validate_json(*expected_args):
    def decorator_validate_json(func):
        @functools.wraps(func)
        def wrapper_validate_json(*args, **kwargs):
            json_object = request.get_json()
            for expected_arg in expected_args:
                if expected_arg not in json_object:
                    abort(400)
            return func(*args, **kwargs)
        return wrapper_validate_json
    return decorator_validate_json
```

In the above code, the decorator takes a variable-length list as an argument so that you can pass in as many string arguments as necessary, each representing a key used to validate the JSON data:

- **Line 4:** The list of keys that must be present in the JSON is given as arguments to the decorator.
- **Line 9:** The wrapper function validates that each expected key is present in the JSON data.

The route handler can then focus on its real job—updating grades—as it can safely assume that the JSON data are valid:

Python decorator_flask.py

```python
import functools
from flask import Flask, request, abort

app = Flask(__name__)

# ...

@app.route("/grade", methods=["POST"])
@validate_json("student_id")
def update_grade():
    json_data = request.get_json()
    # Update database.
    return "success!"
```

You apply `@validate_json`, which simplifies the logic inside `update_grade()`.

Conclusion

This has been quite a journey! You started this tutorial by looking closer at functions, and particularly how you can define them inside other functions and pass them around just like any other Python object. Then you learned about decorators and how to write them such that:

- They can be reused.
- They can decorate functions with arguments and return values.
- They can use `@functools.wraps` to look more like the decorated function.

In the second part of the tutorial, you saw more advanced decorators and learned how to:

- Decorate classes
- Nest decorators
- Add arguments to decorators
- Keep state within decorators
- Use classes as decorators

You saw that, to define a decorator, you typically define a function returning a wrapper function. The wrapper function uses `*args` and `**kwargs` to pass on arguments to the decorated function. If you want your decorator to also take arguments, then you need to nest the wrapper function inside another function. In this case, you usually end up with three `return` statements.

You can download the code from this tutorial by clicking below:

Get Your Code: Click here to download the free sample code that shows you how to create and use Python decorators.

Further Reading

If you're still looking for more, the book *Python Tricks* has a section on decorators, as does the *Python Cookbook* by David Beazley and Brian K. Jones.

For a deep dive into the historical discussion on how decorators should be implemented in Python, see PEP 318 as well as the Python Decorator Wiki. You can find more examples of decorators in the Python Decorator Library. The `decorator` module can simplify creating your own decorators, and its documentation contains further decorator examples.

Decorators Cheat Sheet: Click here to get access to a free three-page Python decorators cheat sheet that summarizes the techniques explained in this tutorial.

Frequently Asked Questions

Now that you have some experience with Python decorators, you can use the questions and answers below to check your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

What are Python decorators and how do they work? Show/Hide

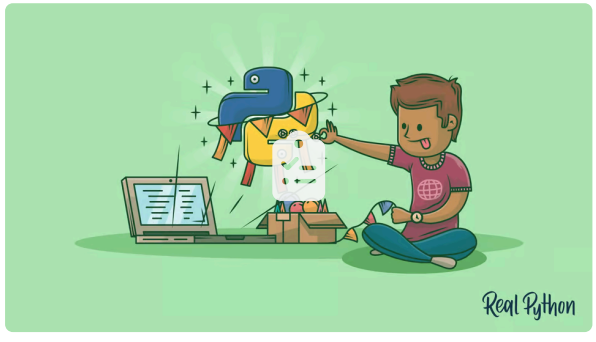
What are some practical use cases for decorators in Python? Show/Hide

How do you write custom decorators in Python? Show/Hide

How do you apply multiple decorators to a single function in Python? Show/Hide

Does order of decorators matter in Python? Show/Hide

Take the Quiz: Test your knowledge with our interactive "Decorators" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Decorators

In this quiz, you'll revisit the foundational concepts of what Python decorators are and how to create and use them.

Mark as Completed Share

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Python Decorators 101**

Python Tricks 💌

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Geir Arne Hjelle**

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

» More about Geir Arne

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Brad

Brenda

Bartosz

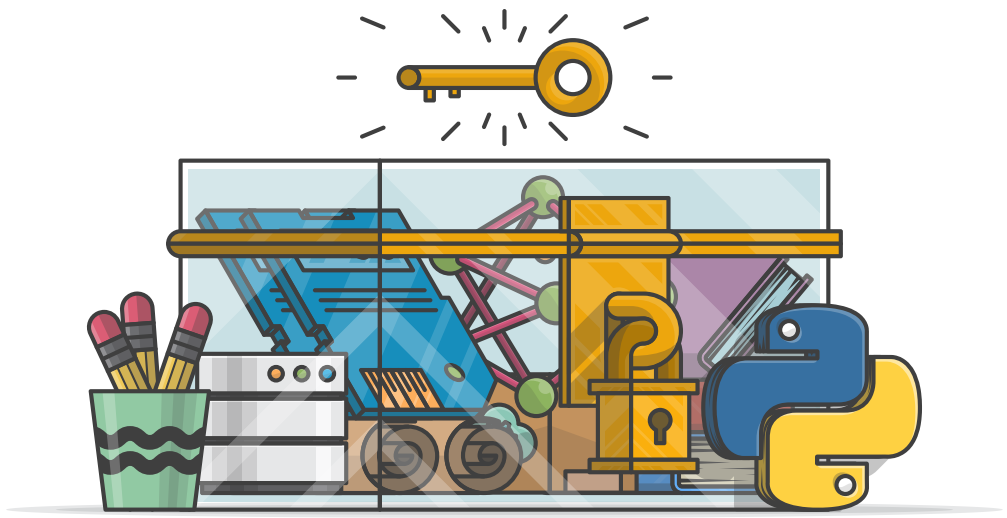
Dan

Joanna

Kate

Michael

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: intermediate python

Recommended Video Course: Python Decorators 101

Related Tutorials:

- Python's property(): Add Managed Attributes to Your Classes
- Python Classes: The Power of Object-Oriented Programming
- Context Managers and Python's with Statement
- Async IO in Python: A Complete Walkthrough

- Python args and kwargs: Demystified

Your **Practical Introduction to Python 3** »

Remove ads