GAMAKA

Artificial Intelligence Solutions

Decorators

Compiled by Mahesh Kaneri

Python Decorators

In Python, functions are the first class objects, which means that – Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.

Functions can be defined inside another function and can also be passed as argument to another function.

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Functions
Before you can understand decorators, you must first understand how functions work. For our purposes, **a function returns a value based on the given arguments**. Here is a very simple example:

```
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 a basic example of this: it returns `None` while having the side effect of outputting something to the console. However, to understand decorators, it is enough to think about functions as something that turns given arguments into a value.

Note: In functional programming, you work (almost) only with pure functions without side effects. While not a purely functional language, Python supports many of the functional programming concepts, including functions as first-class objects.

First-Class Objects

In Python, functions are first-class objects. This means that **functions can be passed around and used as arguments**, just like any other object (string, int, float, list, and so on). Consider the following three functions:

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

def be_awesome(name):
    return f"Yo {name}, together we are 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. We can, for instance, pass it the `say_hello()` or the `be_awesome()` function:

```
 greet_bob(say_hello)
'Hello Bob'

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

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

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:

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

    def first_child():
```

```
        print("Printing from the first_child() function")

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

    second_child()
    first_child()
```

What happens when you call the `parent()` function? Think about this for a minute. The output will be as follows:

```
parent()
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

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 are not defined until the parent function is called. They are locally scoped to `parent()`: they only exist inside the `parent()` function as local variables. Try calling `first_child()`. You should get an error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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.

Returning Functions From Functions

Python also allows you to use functions as return values. The following example returns one of the inner functions from the outer `parent()` function:

```
def parent(num):
    def first_child():
```

```
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

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

Note that you are returning `first_child` without the parentheses. Recall that this means that you are **returning a reference to the function `first_child`**. In contrast `first_child()`with parentheses refers to the result of evaluating the function. This can be seen in the following example:

```
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 simply 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 are regular functions, even though the functions they point to can't be accessed directly:

```
first()
'Hi, I am Emma'

second()
'Call me Liam'
```

Finally, note that in the earlier example you executed the inner functions within the parent function, for instance `first_child()`. However, in this last example, you did not add parentheses to the inner functions—`first_child`—upon

returning. That way, you got a reference to each function that you could call in the future. Make sense?

Simple Decorators

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. Let's start with an example:

```
def my_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 = my_decorator(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 previous examples. We are literally just applying everything you have learned so far.

The so-called decoration happens at the following line:

```
say_whee = my_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 `my_decorator(say_whee)`:

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

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

Put simply: **decorators wrap a function, modifying its behavior.**

Before moving on, let's 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:

```
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:

```
say_whee()
```

Syntactic Sugar!

The way you decorated `say_whee()` above is a little clunky. First of all, you end up typing the name `say_whee` three times. In addition, the decoration gets a bit 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:

```
def my_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

@my_decorator
```

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

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

Reusing Decorators

Recall that a decorator is just a regular Python function. All the usual tools for easy reusability are available. Let's move the decorator to its own module that can be used in many other functions.

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

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

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 article. To keep them apart, we'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:

```
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:

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

Decorating Functions with Arguments

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

```
from decorators import do_twice

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

Unfortunately, running this code raises an error:

```
greet("World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

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

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

```
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 it decorates. Now both your `say_whee()` and `greet()` examples works:

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

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

Returning Values From Decorated Functions

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

```
from decorators import do_twice

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

Try to use it:

```
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")` ended 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:

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

The return value from the last execution of the function is returned:

```
 return_greeting("Adam")
Creating greeting
```

```
Creating greeting
'Hi Adam'
```

Who Are You, Really?

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:

```
 print
<built-in function print>

 print.__name__
'print'

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

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

The introspection works for functions you define yourself as well:

```
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 is not 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:

```
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 do not need to change anything about the decorated `say_whee()` function:

```
 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.

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

A Few Real World Examples

Let's 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:

```
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.

Note: In later examples, we will assume that these decorators are saved in your `decorators.py` file as well. Recall that you can download all the examples in this tutorial.

Timing Functions

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

```
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

This decorator works by storing the time just before the function starts running (at the line marked `# 1`) and just after the function finishes (at `# 2`). The time the function takes is then the difference between the two (at `# 3`). We use the `time.perf_counter()` function, which does a good job of measuring time intervals. Here are some examples of timings:

```
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 code line by line. Make sure you understand how it works. Don't worry if you don't get it, 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, 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 quick function calls.

Debugging Code

The following `@debug` decorator will print the arguments a function is called with as well as its return value every time the function is called:

```
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]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")           # 4
        return value
    return wrapper_debug
```

The signature is created by joining the string representations of all the arguments. The numbers in the following list correspond to the numbered comments in the code:

1. Create a list of the positional arguments. Use `repr()` to get a nice string representing each argument.
2. Create a list of the keyword arguments. The f-string formats each argument as `key=value` where the `!r` specifier means that `repr()` is used to represent the value.

3. The lists of positional and keyword arguments is joined together to one signature string with each argument separated by a comma.
4. The return value is printed after the function is executed.

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

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

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

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

 make_greeting("Richard", age=112)
Calling make_greeting('Richard', age=112)
'make_greeting' returned 'Whoa Richard! 112 already, you are growing up!'
'Whoa Richard! 112 already, you are growing up!'

 make_greeting(name="Dorrisile", age=116)
Calling make_greeting(name='Dorrisile', age=116)
'make_greeting' returned 'Whoa Dorrisile! 116 already, you are growing
up!'
'Whoa Dorrisile! 116 already, you are growing up!'
```

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

The following example calculates an approximation to the mathematical constant *e*:

```
import math
```

```
from decorators import debug

# Apply a decorator to a standard library function
math.factorial = debug(math.factorial)

def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))
```

This example also shows how you can apply a decorator to a function that has already been defined. 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:

```
 approximate_e(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 to the true value *e* = 2.718281828, adding only 5 terms.

Slowing Down Code

This next example 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:

```
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

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

To see the effect of the `@slow_down` decorator, you really need to run the example yourself:

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

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 our guide on 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 they're decorating. They can also simply register that a function exists and return it unwrapped. This can be used, for instance, to create a light-weight plug-in architecture:

```
import random
```

```
PLUGINS = dict()

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

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

@register
def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

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

The `@register` decorator simply stores a reference to the decorated function in the global `PLUGINS` dict. Note that you do not have to write an inner function or use `@functools.wraps` in this example because you are returning the original function unmodified.

The `randomly_greet()` function randomly chooses one of the registered functions to use. Note that the `PLUGINS` dictionary already contains references to each function object that is registered as a plugin:

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

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

The main benefit of this simple plugin architecture is that you do not 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 are familiar with `globals()` in Python, you might see some similarities to how the plugin architecture works. `globals()` gives access to all global variables in the current scope, including your plugins:

```
 globals()
{..., # Lots of variables 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 variables, effectively hand-picking some functions from `globals()`.

Is the User Logged In?

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

```
from flask import Flask, g, request, redirect, url_for
import functools
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 article to practice everything you've learned.

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

- Decorators on classes
- Several decorators on one function
- Decorators with arguments
- Decorators that can optionally take arguments
- Stateful decorators
- Classes as decorators

Decorating Classes

There are two different ways you can use decorators on classes. The first one is very close to what you have 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 that are even built-ins in Python are `@classmethod`, `@staticmethod`, and `@property`.
The `@classmethod` and `@staticmethod` decorators are used to define methods inside a class namespace that are not 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 decoratorsShow/Hide

Let's define a class where we decorate some of its methods using the `@debug` and `@timer`decorators from earlier:

```
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([i**2 for i in range(self.max_num)])
```

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

```
 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
```

The other way to use decorators on classes is to **decorate the whole class**. This is, for example, done in the new `dataclasses` module in Python 3.7:

```
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 have done the decoration 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 are 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 you saw above will work as class decorators. When you are 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:

```
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 does not decorate its methods. Recall that `@timer` is just shorthand for `TimeWaster = timer(TimeWaster)`.

Here, `@timer` only measures the time it takes to instantiate the class:

```
 tw = TimeWaster(1000)
Finished 'TimeWaster' in 0.0000 secs

 tw.waste_time(999)
```

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

Nesting Decorators

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

```
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 are listed. In other words, `@debug` calls `@do_twice`, which calls `greet()`, or `debug(do_twice(greet()))`:

```
 greet("Eva")
Calling greet('Eva')
Hello Eva
Hello Eva
'greet' returned None
```

Observe the difference if we change the order of `@debug` and `@do_twice`:

```
from decorators import debug, do_twice

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

In this case, `@do_twice` will be applied to `@debug` as well:

```
 greet("Eva")
Calling greet('Eva')
Hello Eva
'greet' returned None
Calling greet('Eva')
Hello Eva
'greet' returned None
```

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.

This would allow you to do something like this:

```
@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 could achieve this.

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:

```
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 movie, we'll untangle it all in a moment:

```
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 we have only put the same decorator pattern you have seen many times by now inside one additional `def` that handles the arguments to the decorator. Let's start with the innermost function:

```
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 you have seen, except that it is using the `num_times` parameter that must be supplied from the outside.

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

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

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

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

```
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, we used `repeat` without parentheses to refer to the function object. The added parentheses are necessary when defining decorators that take 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 it will be used later by `wrapper_repeat()`.

With everything set up, let's see if the results are as expected:

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

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

Just the result we were aiming for.

Both Please, But Never Mind the Bread

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.

As you saw in the previous section, when a decorator uses arguments, you need to add an extra outer function. The challenge is for your code to figure out if the decorator has been called 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 `*` syntax, which means that all following parameters are keyword-only:

```
def name(_func=None, *, kw1=val1, kw2=val2, ...):  # 1
    def decorator_name(func):
        ...  # Create and return a wrapper function.

    if _func is None:
        return decorator_name                      # 2
    else:
        return decorator_name(_func)               # 3
```

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

1. If `name` has been called without arguments, the decorated function will be passed in as `_func`. If it has been called with arguments, then `_func` will be `None`, and some of the keyword arguments may have been changed from their default values. The `*` in the argument list means that the remaining arguments can't be called as positional arguments.
2. In this case, the decorator was called with arguments. Return a decorator function that can read and return a function.

3. In this case, the decorator was called 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:

```
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` at the end.

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

These examples show that `@repeat` can now be used with or without arguments:

```
@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:

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

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

Stateful Decorators

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

Note: In the beginning of this guide, we talked 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 will see how to use classes to keep state. But in simple cases, you can also get away with using function attributes:

```
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__!r}")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls

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

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

```
 say_whee()
Call 1 of 'say_whee'
Whee!
```

```
 say_whee()
Call 2 of 'say_whee'
Whee!

 say_whee.num_calls
2
```

Classes as Decorators

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

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

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

```
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:

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

 counter()
Current count is 2

 counter.count
```

```
2
```

Therefore, a typical implementation of a decorator class needs to implement `.__init__()`and `.__call__()`:

```
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__!r}")
        return self.func(*args, **kwargs)

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

The `.__init__()` method must store a reference to the function and 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 our 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:

```
 say_whee()
Call 1 of 'say_whee'
Whee!

 say_whee()
Call 2 of 'say_whee'
Whee!

 say_whee.num_calls
2
```

More Real World Examples

We've come a far way now, having figured out how to create all kinds of decorators. Let's wrap it up, putting our newfound knowledge into creating a few more examples that might actually be useful in the real world.

Slowing Down Code, Revisited

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

```
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)
```

We're using the boilerplate introduced in the Both Please, But Never Mind the Bread 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:

```
@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:

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

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`. It is the fact that `None` is a singleton that allows you to compare for `None` using the `is` keyword, like you saw in the Both Pleasesection:

```
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:

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton

@singleton
class TheOne:
    pass
```

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

Let's see if it works:

```
first_one = TheOne()
another_one = TheOne()

id(first_one)
140094218762280

id(another_one)
140094218762280

first_one is another_one
True
```

It seems clear that `first_one` is indeed the exact same instance as `another_one`.

Note: Singleton classes are not really used as often in Python as in other languages. The effect of a singleton is usually better implemented as a global variable in a module.

Caching Return Values

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

```
from decorators import count_calls

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

While the implementation is simple, its runtime performance is terrible:

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

fibonacci.num_calls
```

```
177
```

To calculate the tenth Fibonacci number, you should really only need to calculate the preceding Fibonacci numbers, but this implementation somehow needs a whopping 177 calculations. It gets worse quickly: 21891 calculations are needed for `fibonacci(20)` and almost 2.7 million calculations for the 30th 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, simple caching of the calculations will also do the trick:

```
import functools
from decorators import count_calls

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 = dict()
    return wrapper_cache

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

The cache works as a lookup table, so now `fibonacci()` only does the necessary calculations once:

```
 fibonacci(10)
Call 1 of 'fibonacci'
...
Call 11 of 'fibonacci'
```

```
55
```

```
fibonacci(8)
```

```
21
```

Note that in the final 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`.

This decorator has more features than the one you saw above. You should use `@functools.lru_cache` instead of writing your own cache decorator:

```
import functools

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

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. However, be aware that this can cause memory problems if you are 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 our example, we used an artificially small `maxsize` to see the effect of elements being removed from the cache:

```
fibonacci(10)
```

```
Calculating fibonacci(10)
Calculating fibonacci(9)
Calculating fibonacci(8)
Calculating fibonacci(7)
Calculating fibonacci(6)
```

```
Calculating fibonacci(5)
Calculating fibonacci(4)
Calculating fibonacci(3)
Calculating fibonacci(2)
Calculating fibonacci(1)
Calculating fibonacci(0)
55

 fibonacci(8)
21

 fibonacci(5)
Calculating fibonacci(5)
Calculating fibonacci(4)
Calculating fibonacci(3)
Calculating fibonacci(2)
Calculating fibonacci(1)
Calculating fibonacci(0)
5

 fibonacci(8)
Calculating fibonacci(8)
Calculating fibonacci(7)
Calculating fibonacci(6)
21

 fibonacci(5)
5

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

Adding Information About Units

The following example is somewhat similar to the Registering Plugins example from earlier, in that it does not really change the behavior of the decorated function. Instead, it simply adds `unit` as a function attribute:

```
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:

```
import math

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

This `.unit` function attribute can later be accessed when needed:

```
 volume(3, 5)
141.3716694115407

 volume.unit
'cm^3'
```

Note that you could have achieved something similar using function annotations:

```
import math

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

However, since annotations are used for type hints, it would be hard 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 (`pip install Pint`), you can for instance convert the volume to cubic inches or gallons:

```
 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 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`. The registry is stored as a function attribute to avoid cluttering the namespace:

```
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

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

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

```
 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
```

Validating JSON

Let's look at one last use case. Take a quick look at the following Flask route handler:

```
@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 we ensure that the key `student_id` is part of the request. Although this validation works, it really does not belong in the function itself. Plus, perhaps there are other routes that use the exact same validation. So, let's keep it DRY and abstract out any unnecessary logic with a decorator. The following `@validate_json` decorator will do the job:

```
from flask import Flask, request, abort
import functools
app = Flask(__name__)

def validate_json(*expected_args):                  # 1
    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:      # 2
                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 we can pass in as many string arguments as necessary, each representing a key used to validate the JSON data:

1. The list of keys that must be present in the JSON is given as arguments to the decorator.

2. 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 JSON data are valid:

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

Conclusion

This has been quite a journey! You started this tutorial by looking a little closer at functions, particularly how they can be defined inside other functions and passed 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, you need to nest the wrapper function inside another function. In this case, you usually end up with three `return`statements.