Mastering Python Decorators

Aaron Maxwell

aaron@powerfulpython.com

Contents

Chapter 1

Advanced Functions

In this chapter, we go beyond the basics of using functions. I'll assume you can define and work with functions taking default arguments:

```
>>> def foo(a, b, x=3, y=2):
...     return (a+b)/(x+y)
...
>>> foo(5, 0)
1.0
>>> foo(10, 2, y=3)
2.0
>>> foo(b=4, x=8, a=1)
0.5
```

Notice the last way `foo` is called: with the arguments out of order, and everything specified by key-value pairs. Not everyone knows that you can call *any* function in Python this way. So long as the value of each argument is unambiguously specified, Python doesn't care how you call the function (and this case, we specify `b`, `x` and `a` out of order, letting `y` be its default value). We'll leverage this flexibility later.

This chapter's topics are useful and valuable on their own. And they are important building blocks for some *extremely* powerful patterns, which you learn in later chapters. Let's get started!

1.1 Accepting & Passing Variable Arguments

The `foo` function above can be called with either 2, 3, or 4 arguments. Sometimes you want to define a function that can take *any* number of arguments - zero or more, in other words. In Python, it looks like this:

```
# Note the asterisk. That's the magic part
def takes_any_args(*args):
    print("Type of args: " + str(type(args)))
    print("Value of args: " + str(args))
```

See carefully the syntax here. `takes_any_args` is just like a regular function, except you put an asterisk right before the argument `args`. Within the function, `args` is a tuple:

```
>>> takes_any_args("x", "y", "z")
Type of args: <class 'tuple'>
Value of args: ('x', 'y', 'z')
>>> takes_any_args(1)
Type of args: <class 'tuple'>
Value of args: (1,)
>>> takes_any_args()
Type of args: <class 'tuple'>
Value of args: ()
>>> takes_any_args(5, 4, 3, 2, 1)
Type of args: <class 'tuple'>
Value of args: (5, 4, 3, 2, 1)
>>> takes_any_args(["first", "list"], ["another","list"])
Type of args: <class 'tuple'>
Value of args: (['first', 'list'], ['another', 'list'])
```

If you call the function with no arguments, `args` is an empty tuple. Otherwise, it is a tuple composed of those arguments passed, in order. This is different from declaring a function that takes a single argument, which happens to be of type list or tuple:

```
>>> def takes_a_list(items):
...     print("Type of items: " + str(type(items)))
...     print("Value of items: " + str(items))
...
>>> takes_a_list(["x", "y", "z"])
Type of items: <class 'list'>
Value of items: ['x', 'y', 'z']
>>> takes_any_args(["x", "y", "z"])
Type of args: <class 'tuple'>
Value of args: (['x', 'y', 'z'],)
```

In these calls to `takes_a_list` and `takes_any_args`, the argument `items` is a list of strings. We're calling both functions the exact same way, but what happens in each function is different. Within `takes_any_args`, the tuple named `args` has one element - and that element is the list `["x", "y", "z"]`. But in `takes_a_list`, `items` is the list itself.

This `*args` idiom gives you some *very* helpful programming patterns. You can work with arguments as an abstract sequence, while providing a potentially more natural interface for whomever calls the function.

Above, I've always named the argument `args` in the function signature. Writing `*args` is a well-followed convention, but you can choose a different name - the asterisk is what makes it a variable argument. For instance, this takes paths of several files as arguments:

```
def read_files(*paths):
    data = ""
    for path in paths:
        with open(path) as handle:
            data += handle.read()
    return data
```

Most Python programmers use `*args` unless there is a reason to name it something else.[1] That reason is usually readability; `read_files` is a good example. If naming it something other than `args` makes the code more understandable, do it.

[1]This seems to be deeply ingrained; once I abbreviated it `*a`, only to have my code reviewer demand I change it to `*args`. They wouldn't approve it until I changed it, so I did.

Argument Unpacking

The star modifier works in the other direction too. Intriguingly, you can use it with *any* function. For example, suppose a library provides this function:

```
def order_book(title, author, isbn):
    """
    Place an order for a book.
    """
    print("Ordering '{}' by {} ({})".format(
        title, author, isbn))
    # ...
```

Notice there's no asterisk. Suppose in another, completely different library, you fetch the book info from this function:

```
def get_required_textbook(class_id):
    """
    Returns a tuple (title, author, ISBN)
    """
    # ...
```

Again, no asterisk. Now, one way you can bridge these two functions is to store the tuple result from `get_required_textbook`, then unpack it element by element:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(book_info[0], book_info[1], book_info[2])
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

Writing code this way is tedious and error-prone; not ideal.

Fortunately, Python provides a better way. Let's look at a different function:

```
def normal_function(a, b, c):
    print("a: {} b: {} c: {}".format(a,b,c))
```

No trick here - it really is a normal, boring function, taking three arguments. If we have those three arguments as a list or tuple, Python can automatically "unpack" them for us. We just need to pass in that collection, prefixed with an asterisk:

```
>>> numbers = (7, 5, 3)
>>> normal_function(*numbers)
a: 7 b: 5 c: 3
```

Again, `normal_function` is just a regular function. We did not use an asterisk on the `def` line. But when we call it, we take a tuple called `numbers`, and pass it in with the asterisk in front. This is then unpacked *within the function* to the arguments `a`, `b`, and `c`.

There is a duality here. We can use the asterisk syntax both in *defining* a function, and in *calling* a function. The syntax looks very similar. But realize they are doing two different things. One is packing arguments into a tuple automatically - called "variable arguments"; the other is *un*-packing them - called "argument unpacking". Be clear on the distinction between the two in your mind.

Armed with this complete understanding, we can bridge the two book functions in a much better way:

```
>>> book_info = get_required_textbook(4242)
>>> order_book(*book_info)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

This is more concise (less tedious to type), and more maintainable. As you get used to the concepts, you'll find it increasingly natural and easy to use in the code you write.

Variable Keyword Arguments

So far we have just looked at functions with *positional* arguments - the kind where you declare a function like `def foo(a, b):`, and then invoke it like `foo(7, 2)`. You know that a=7 and b=2 within the function, because of the order of the arguments. Of course, Python also has keyword arguments:

```
>>> def get_rental_cars(size, doors=4,
...         transmission='automatic'):
...     template = "Looking for a {}-door {} car with {}
   transmission...."
...     print(template.format(doors, size, transmission))
...
>>> get_rental_cars("economy", transmission='manual')
Looking for a 4-door economy car with manual transmission....
```

And remember, Python lets you call *any* function just using keyword arguments:

```
>>> def bar(x, y, z):
...     return x + y * z
...
>>> bar(z=2, y=3, x=4)
10
```

These keyword arguments won't be captured by the `*args` idiom. Instead, Python provides a different syntax - using two asterisks instead of one:

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
        print("{} -> {}".format(key, value))
```

The variable `kwargs` is a *dictionary*. (In contrast to `args` - remember, that was a tuple.) It's just a regular `dict`, so we can iterate through its key-value pairs with `.items()`:[2]

```
>>> print_kwargs(hero="Homer", antihero="Bart",
...     genius="Lisa")
hero -> Homer
antihero -> Bart
genius -> Lisa
```

The arguments to `print_kwargs` are key-value pairs. This is regular Python syntax for calling functions; what's interesting is happening *inside* the function. There, a variable called `kwargs` is defined. It's a Python dictionary, consisting of the key-value pairs passed in when the function was called.

Here's another example, which has a regular positional argument, followed by arbitrary key-value pairs:

```
def set_config_defaults(config, **kwargs):
    for key, value in kwargs.items():
        # Do not overwrite existing values.
        if key not in config:
            config[key] = value
```

This is perfectly valid. You can define a function that takes some normal arguments, followed by zero or more key-value pairs:

[2] Or `.viewitems()`, if you're using Python 2.

```
>>> config = {"verbosity": 3, "theme": "Blue Steel"}
>>> set_config_defaults(config, bass=11, verbosity=2)
>>> config
{'verbosity': 3, 'theme': 'Blue Steel', 'bass': 11}
```

Like with `*args`, naming this variable `kwargs` is just a strong convention; you can choose a different name if that improves readability.

Keyword Unpacking

Just like with `*args`, double-star works the other way too. We can take a regular function, and pass it a dictionary using two asterisks:

```
>>> def normal_function(a, b, c):
...     print("a: {} b: {} c: {}".format(a,b,c))
...
>>> numbers = {"a": 7, "b": 5, "c": 3}
>>> normal_function(**numbers)
a: 7 b: 5 c: 3
```

Note the keys of the dictionary *must* match up with how the function was declared. Otherwise you get an error:

```
>>> bad_numbers = {"a": 7, "b": 5, "z": 3}
>>> normal_function(**bad_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: normal_function() got an unexpected keyword argument 'z'
```

This is called *keyword argument unpacking*. It works regardless of whether that function has default values for some of its arguments or not. So long as the value of each argument is specified one way or another, you have valid code:

```
>>> def another_function(x, y, z=2):
...     print("x: {} y: {} z: {}".format(x,y,z))
...
>>> all_numbers = {"x": 2, "y": 7, "z": 10}
>>> some_numbers = {"x": 2, "y": 7}
>>> missing_numbers = {"x": 2}
>>> another_function(**all_numbers)
x: 2 y: 7 z: 10
>>> another_function(**some_numbers)
x: 2 y: 7 z: 2
>>> another_function(**missing_numbers)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: another_function() missing 1 required positional argument
   : 'y'
```

Combining Positional and Keyword Arguments

You can combine the syntax to use both positional and keyword arguments. In a function signature, just separate `*args` and `**kwargs` by a comma:

```
>>> def general_function(*args, **kwargs):
...     for arg in args:
...         print(arg)
...     for key, value in kwargs.items():
...         print("{} -> {}".format(key, value))
...
>>> general_function("foo", "bar", x=7, y=33)
foo
bar
y -> 33
x -> 7
```

This usage - declaring a function like `def general_function(*args, **kwargs)` - is the most general way to define a function in Python. A function so declared can be called in any way, with any valid combination of keyword and non-keyword arguments - including no arguments.

Similarly, you can call a function using both - and both will be unpacked:

```
>>> def addup(a, b, c=1, d=2, e=3):
...     return a + b + c + d + e
...
>>> nums = (3, 4)
>>> extras = {"d": 5, "e": 2}
>>> addup(*nums, **extras)
15
```

There's one last point to understand, on argument ordering. When you `def` the function, you specify the arguments in this order:

- Named, regular (non-keyword) arguments, then
- the `*args` non-keyword variable arguments, then
- the `**kwargs` keyword variable arguments, and finally
- required keyword-only arguments.

You can omit any of these when defining a function. But any that are present *must* be in this order.

```
# All these are valid function definitions.
def combined1(a, b, *args): pass
def combined2(x, y, z, **kwargs): pass
def combined3(*args, **kwargs): pass
def combined4(x, *args): pass
def combined5(u, v, w, *args, **kwargs): pass
def combined6(*args, x, y): pass
```

Violating this order will cause errors:

```
>>> def bad_combo(**kwargs, *args): pass
  File "<stdin>", line 1
    def bad_combo(**kwargs, *args): pass
                          ^
SyntaxError: invalid syntax
```

Sometimes you might want to define a function that takes 0 or more positional arguments, and 1 or more *required* keyword arguments. You can define a function like this with `*args` followed by regular arguments, forming a special category, called *keyword-only arguments*.[3] If present, whenever that function is called, all must specified as key-value pairs, *after* the non-keyword arguments:

```
>>> def read_data_from_files(*paths, format):
...     """Read and merge data from several files,
...     which are in XML, JSON, or YAML format."""
...     # ...
...
>>> housing_files = ["houses.json", "condos.json"]
>>> housing_data = read_data_from_files(
...     *housing_files, format="json")
>>> commodities_data = read_data_from_files(
        "commodities.xml", format="xml")
```

See how `format`'s value is specified with a key-value pair. If you try passing it without `format=` in front, you get an error:

```
>>> commodities_data = read_data_from_files(
...     "commodities.xml", "xml")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: read_data_from_files() missing 1 required keyword-only
   argument: 'format'
```

1.2 Functions As Objects

In Python, functions are ordinary objects - just like an integer, a list, or an instance of a class you create. The implications are profound, letting you do certain *very* useful things with functions. Leveraging this is one of those secrets separating average Python developers from great ones, because of the *extremely* powerful abstractions which follow.

Once you get this, it can change the way you write software forever. In fact, these advanced patterns for using functions in Python largely transfer to other languages you will use in the future.

[3] These are newly available in Python 3. For Python 2, it's an error to define a function with any regular arguments after `*args`.

To explain, let's start by laying out a problematic situation, and how to solve it. Imagine you have a list of strings representing numbers:

```
nums = ["12", "7", "30", "14", "3"]
```

Suppose we want to find the biggest integer in this list. The `max` builtin does not help us:

```
>>> max(nums)
'7'
```

This isn't a bug, of course; since the objects in `nums` are strings, `max` compares each element lexicographically.[4] By that criteria, "7" is greater than "30", for the same reason "g" comes after "ca" alphabetically. Essentially, `max` is evaluating the element by a different criteria than what we want.

Since `max`'s algorithm is simple, let's roll our own that compares based on the integer value of the string:

```
>>> def max_by_int_value(items):
...     # For simplicity, assume len(items) > 0
...     biggest = items[0]
...     for item in items[1:]:
...         if int(item) > int(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_int_value(nums)
'30'
```

This gives us what we want: it returns the element in the original list which is maximal, as evaluated by our criteria. Now imagine working with different data, where you have different criteria. For example, a list of actual integers:

```
integers = [3, -2, 7, -1, -20]
```

Suppose we want to find the number with the greatest *absolute value* - i.e., distance from zero. That would be -20 here, but standard `max` won't do that:

```
>>> max(integers)
7
```

[4]Meaning, alphabetically, but generalizing beyond the letters of the alphabet.

Again, let's roll our own, using the built-in abs function:

```
>>> def max_by_abs(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if abs(item) > abs(biggest):
...             biggest = item
...     return biggest
...
>>> max_by_abs(integers)
-20
```

One more example - a list of dictionary objects:

```
student_joe = {'gpa': 3.7, 'major': 'physics',
               'name': 'Joe Smith'}
student_jane = {'gpa': 3.8, 'major': 'chemistry',
                'name': 'Jane Jones'}
student_zoe = {'gpa': 3.4, 'major': 'literature',
               'name': 'Zoe Fox'}
students = [student_joe, student_jane, student_zoe]
```

Now, what if we want the record of the student with the highest GPA? Here's a suitable max function:

```
>>> def max_by_gpa(items):
...     biggest = items[0]
...     for item in items[1:]:
...         if item["gpa"] > biggest["gpa"]:
...             biggest = item
...     return biggest
...
>>> max_by_gpa(students)
{'name': 'Jane Jones', 'gpa': 3.8, 'major': 'chemistry'}
```

Just one line of code is different between `max_by_int_value`, `max_by_abs`, and `max_by_gpa`: the comparison line. `max_by_int_value` says `if int(item) > int(biggest)`; `max_by_abs` says `if abs(item) > abs(biggest)`; and `max_by_gpa` compares `item["gpa"]` to `biggest["gpa"]`. Other than that, these `max` functions are identical.

I don't know about you, but having nearly-identical functions like this drives me nuts. The way out is to realize the comparison is based on a value *derived* from the element - not the value of the element itself. In other words: each cycle through the for loop, the two elements are **not** themselves compared. What is compared is some derived, calculated value: `int(item)`, or `abs(item)`, or `item["gpa"]`.

It turns out we can abstract out that calculation, using what we'll call a *key function*. A key function is a function that takes exactly one argument - an element in the list. It returns the derived value used in the comparison. In fact, `int` works like a function, even though it's technically a type, because `int("42")` returns `42`.[5] So types and other callables work, as long as we can invoke it like a one-argument function.

This lets us define a very generic max function:

```
>>> def max_by_key(items, key):
...     biggest = items[0]
...     for item in items[1:]:
...         if key(item) > key(biggest):
...             biggest = item
...     return biggest
...
>>> # Old way:
... max_by_int_value(nums)
'30'
>>> # New way:
... max_by_key(nums, int)
'30'
>>> # Old way:
... max_by_abs(integers)
-20
>>> # New way:
... max_by_key(integers, abs)
-20
```

Pay attention: you are passing the function object itself - `int` and `abs`. You are *not* invoking the key function in any direct way. In other words, you write `int`, not `int()`. This function object is then called as needed by `max_by_key`, to calculate the derived value:

```
        # key is actually int, abs, etc.
        if key(item) > key(biggest):
```

[5]Python uses the word *callable* to describe something that can be invoked like a function. This can be an actual function, a type or class name, or an object defining the `__call__` magic method. Key functions are frequently actual functions, but can be any callable.

For sorting the students by GPA, we need a function extracting the "gpa" key from each student dictionary. There is no built-in function that does this, but we can define our own and pass it in:

```
>>> # Old way:
... max_by_gpa(students)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}

>>> # New way:
... def get_gpa(who):
...     return who["gpa"]
...
>>> max_by_key(students, get_gpa)
{'gpa': 3.8, 'name': 'Jane Jones', 'major': 'chemistry'}
```

Again, notice `get_gpa` is a function object, and we are passing that function itself to `max_by_key`. We never invoke `get_gpa` directly; `max_by_key` does that automatically.

You may be realizing now just how powerful this can be. In Python, functions are simply objects - just as much as an integer, or a string, or an instance of a class is an object. You can store functions in variables; pass them as arguments to other functions; and even return them from other function and method calls. This all provides new ways for you to encapsulate and control the behavior of your code.

The Python standard library demonstrates some excellent ways to use such functional patterns. Let's look at a key (ha!) example.

1.3 Key Functions in Python

Earlier, we saw the built-in `max` doesn't magically do what we want when sorting a list of numbers-as-strings:

```
>>> nums = ["12", "7", "30", "14", "3"]
>>> max(nums)
'7'
```

Again, this isn't a bug - `max` just compares elements according to the data type, and `"7" > "12"` evaluates to `True`. But it turns out `max` is customizable. You can pass it a key function!

```
>>> max(nums, key=int)
'30'
```

The value of key is a function taking one argument - an element in the list - and returning a value for comparison. But max isn't the only built-in accepting a key function. min and sorted do as well:

```
>>> # Default behavior...
... min(nums)
'12'
>>> sorted(nums)
['12', '14', '3', '30', '7']
>>>
>>> # And with a key function:
... min(nums, key=int)
'3'
>>> sorted(nums, key=int)
['3', '7', '12', '14', '30']
```

Many algorithms can be cleanly expressed using min, max, or sorted, along with an appropriate key function. Sometimes a built-in (like int or abs) will provide what you need, but often you'll want to create a custom function. Since this is so commonly needed, the operator module provides some helpers. Let's revisit the example of a list of student records.

```
>>> student_joe = {'gpa': 3.7, 'major': 'physics',
        'name': 'Joe Smith'}
>>> student_jane = {'gpa': 3.8, 'major': 'chemistry',
        'name': 'Jane Jones'}
>>> student_zoe = {'gpa': 3.4, 'major': 'literature',
        'name': 'Zoe Fox'}
>>> students = [student_joe, student_jane, student_zoe]
>>>
>>> def get_gpa(who):
...     return who["gpa"]
...
>>> sorted(students, key=get_gpa)
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
```

This is effective, and a fine way to solve the problem. Alternatively, the `operator` module's `itemgetter` creates and returns a key function that looks up a named dictionary field:

```
>>> from operator import itemgetter
>>>
>>> # Sort by GPA...
... sorted(students, key=itemgetter("gpa"))
[{'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'},
 {'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'}]
>>>
>>> # Now sort by major:
... sorted(students, key=itemgetter("major"))
[{'gpa': 3.8, 'major': 'chemistry', 'name': 'Jane Jones'},
 {'gpa': 3.4, 'major': 'literature', 'name': 'Zoe Fox'},
 {'gpa': 3.7, 'major': 'physics', 'name': 'Joe Smith'}]
```

Notice `itemgetter` is a function that creates and returns a function - itself a good example of how to work with function objects. In other words, the following two key functions are completely equivalent:

```
# What we did above:
def get_gpa(who):
    return who["gpa"]

# Using itemgetter instead:
from operator import itemgetter
get_gpa = itemgetter("gpa")
```

This is how you use `itemgetter` when the sequence elements are dictionaries. It also works when the elements are tuples or lists - just pass a number index instead:

```
>>> # Same data, but as a list of tuples.
... student_rows = [
...     ("Joe Smith", "physics", 3.7),
...     ("Jane Jones", "chemistry", 3.8),
...     ("Zoe Fox", "literature", 3.4),
...     ]
>>>
>>> # GPA is the 3rd item in the tuple, i.e. index 2.
... # Highest GPA:
... max(student_rows, key=itemgetter(2))
('Jane Jones', 'chemistry', 3.8)
>>>
>>> # Sort by major:
... sorted(student_rows, key=itemgetter(1))
[('Jane Jones', 'chemistry', 3.8),
 ('Zoe Fox', 'literature', 3.4),
 ('Joe Smith', 'physics', 3.7)]
```

operator also provides `attrgetter`, for keying off an attribute of the element, and `methodcaller` for keying off a method's return value - useful when the sequence elements are instances of your own class:

```
>>> class Student:
...     def __init__(self, name, major, gpa):
...         self.name = name
...         self.major = major
...         self.gpa = gpa
...     def __repr__(self):
...         return "{}: {}".format(self.name, self.gpa)
...
>>> student_objs = [
...     Student("Joe Smith", "physics", 3.7),
...     Student("Jane Jones", "chemistry", 3.8),
...     Student("Zoe Fox", "literature", 3.4),
...     ]
>>> from operator import attrgetter
>>> sorted(student_objs, key=attrgetter("gpa"))
[Zoe Fox: 3.4, Joe Smith: 3.7, Jane Jones: 3.8]
```

Chapter 2

Decorators

Python supports a powerful tool called the *decorator*. Decorators let you add rich features to groups of functions and classes, without modifying them at all; untangle distinct, frustratingly intertwined concerns in your code, in ways not otherwise possible; and build powerful, extensible software frameworks. Many of the most popular and important Python libraries in the world leverage decorators. This chapter teaches you how to do the same.

A decorator is something you apply to a function or method. You've probably seen decorators before. There's a decorator called `property` often used in classes:

```
>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...
...     @property
...     def full_name(self):
...         return self.first_name + " " + self.last_name
...
>>> person = Person("John", "Smith")
>>> print(person.full_name)
John Smith
```

(Note what's printed: `person.full_name`, not `person.full_name()`.) Another example: in the Flask web framework, here is how you define a simple home page:

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

The `app.route("/")` is a decorator, applied here to the function called `hello`. So an HTTP GET request to the root URL ("/") will be handled by the `hello` function.

A decorator works by **adding behavior *around*** a function - meaning, lines of code which are executed before that function begins, after it returns, or both. It does not alter any lines of code *inside* the function. Typically, when you go to the trouble to define a decorator, you plan use it on at least two different functions, usually more. Otherwise you'd just put the extra code inside the lone function, and not bother writing a decorator.

Using decorators is simple and easy; even someone new to programming can learn that quickly. Our objective is different: to give you the ability to *write* your own decorators, in many different useful forms. This is not a beginner topic; it barely qualifies as intermediate. It requires a deep understanding of several sophisticated Python features, and how they play together. Most Python developers never learn how to create them. In this chapter, you will.[1]

2.1 The Basic Decorator

Once a decorator is written, using it is easy. You just write `@` and the decorator name, on the line before you define a function:

```
@some_decorator
def some_function(arg):
    # blah blah
```

This applies the decorator called `some_decorator` to `some_function`.[2] Now, it turns out this syntax with the @ symbol is a shorthand. In essence, when byte-compiling your code, Python will translate the above into this:

```
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

[1]Writing decorators builds on the "Advanced Functions" chapter. If you are not already familiar with that material, read it first.

[2]For Java people: this looks just like Java annotations. However, it's *completely different*. Python decorators are not in any way similar.

This is valid Python code too, and what people did before the `@` syntax came along. The key here is the last line:

```
some_function = some_decorator(some_function)
```

First, understand that **a decorator is just a function**. That's it. It happens to be a function taking one argument, which is the function object being decorated. It then returns a different function. In the code snippet above is you defining a function, initially called `some_function`. That function object is passed to `some_decorator`, which returns a *different* function object, which is finally stored in `some_function`.

To keep us sane, let's define some terminology:

- The **decorator** is what comes after the @. It's a function.
- The **bare function** is what's `def`'ed on the next line. It is, obviously, also a function.
- The end result is the **decorated function**. It's the final function you actually call in your code.[3]

Your mastery of decorators will be most graceful if you remember one thing: a decorator is just a normal, boring function. It happens to be a function taking exactly one argument, which is itself a function. And when called, the decorator returns a *different* function.

Let's make this concrete. Here's a simple decorator which logs a message to stdout, every time the decorated function is called.

```
def printlog(func):
    def wrapper(arg):
        print('CALLING: {}'.format(func.__name__))
        return func(arg)
    return wrapper

@printlog
def foo(x):
    print(x + 2)
```

Notice this decorator creates a new function, called `wrapper`, and returns that. This is then assigned to the variable `foo`, replacing the undecorated, bare function:

[3] Some authors use the phrase "decorated function" to mean "the function that is decorated" - what I'm calling the "bare function". If you read a lot of blog posts, you'll find the phrase used both ways (sometimes in the same article), but we'll consistently use the definitions above.

```
# Remember, this...
@printlog
def foo(x):
    print(x + 2)

# ...is the exact same as this:
def foo(x):
    print(x + 2)
foo = printlog(foo)
```

Here's the result:

```
>>> foo(3)
CALLING: foo
5
```

At a high level, the body of `printlog` does two things: define a function called `wrapper`, then return it. Many decorators will follow that structure. Notice `printlog` does not modify the behavior of the original function `foo` itself; all `wrapper` does is print a message to standard out, before calling the original (bare) function.

Once you've applied a decorator, the bare function isn't directly accessible anymore; you can't call it in your code. Its name now applies to the decorated version. But that decorated function internally retains a reference to the bare function, calling it inside `wrapper`.

This version of `printlog` has a big shortcoming, though. Look what happens when I apply it to a different function:

```
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> baz(3,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper() takes 1 positional argument but 2 were given
```

Can you spot what went wrong?

`printlog` is built to wrap a function taking exactly one argument. But `baz` has two, so when the decorated function is called, the whole thing blows up. There's no reason `printlog` needs to have this restriction; all it's doing is printing the function name. You fix it by declaring `wrapper` with variable arguments:

```
# A MUCH BETTER printlog.
def printlog(func):
    def wrapper(*args, **kwargs):
        print('CALLING: {}'.format(func.__name__))
        return func(*args, **kwargs)
    return wrapper
```

This decorator is compatible with *any* Python function:

```
>>> @printlog
... def foo(x):
...     print(x + 2)
...
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

A decorator written this way, using variable arguments, will potentially work with functions and methods written *years* later - code the original developer never imagined. This structure has proven very powerful and versatile.

```
# The prototypical form of Python decorators.
def prototype_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

We don't always do this, though. Sometimes you are writing a decorator that only applies to a function or method with a very specific kind of signature, and it would be an error to use it anywhere else. So feel free to break this rule when you have a reason.

Decorators apply to methods just as well as to functions. And you often don't need to change anything: when the wrapper has a signature of `wrapper(*args, **kwargs)`, like `printlog` does, it works just fine with any object's method. But sometimes you will see code like this:

```python
# Not really necessary.
def printlog_for_method(func):
    def wrapper(self, *args, **kwargs):
        print('CALLING: {}'.format(func.__name__))
        return func(self, *args, **kwargs)
    return wrapper
```

This is a bit interesting. This `wrapper` has one required argument, named `self`. And it works fine when applied to a method. But for the decorator I've written here, that `self` is completely unnecessary, and in fact has a downside.

Simply defining `wrapper(*args, **kwargs)` causes `self` to be considered one of the `args`; such a decorator works just as well with both functions and methods. But if a wrapper is defined to require `self`, that means it must always be called with at least one argument. Suddenly you have a decorator that cannot be applied to functions without at least one argument. (That it's named `self` doesn't matter; it's just a temporary name for that first argument, inside the scope of `wrapper`.) You can apply this decorator to any method, and to some functions. But if you apply it a function that takes *no* arguments, you'll get a run-time error.

Now, here's a different decorator:

```python
# This is more sensible.
def enhanced_printlog_for_method(func):
    def wrapper(self, *args, **kwargs):
        print('CALLING: {} on object ID {}'.format(
            func.__name__, id(self)))
        return func(self, *args, **kwargs)
    return wrapper
```

It could be applied like this:

```
class Invoice:
    def __init__(self, id_number, total):
        self.id_number = id_number
        self.total = total
        self.owed = total
    @enhanced_printlog_for_method
    def record_payment(self, amount):
        self.owed -= amount

inv = Invoice(42, 117.55)
print("ID of inv: {}".format(id(inv)))
inv.record_payment(55.35)
```

Here's the output when you execute:

```
ID of inv: 4320786472
CALLING: record_payment on object ID 4320786472
```

This is a different story, because this `wrapper`'s body explicitly uses the current object - a concept that only makes sense for methods. That makes the `self` argument perfectly appropriate. It prevents you from using this decorator on some functions, but it would actually be an error to apply it to a non-method anyway.

When writing a decorator for methods, I recommend you get in the habit of making your wrapper only take `*args` and `**kwargs`, except when you have a clear reason to do differently. After you've written decorators for a while, you'll be surprised at how often you end up using old decorators on new functions, in ways you never imagined at first. A signature of `wrapper(*args, **kwargs)` preserves that flexibility. If the decorator turns out to need an explicit `self` argument, it's easy enough to put that in.

2.2 Data In Decorators

Some of the most valuable decorator patterns rely on using variables inside the decorator function itself. This is *not* the same as using variables inside the *wrapper* function. Let me explain.

Imagine you need to keep a running average of what some function returns. And further, you need to do this for a family of functions or methods. We can write a decorator called `running_average` to handle this - as you read, note carefully how `data` is defined and used:

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

Each time the function is called, the average of all calls so far is printed out.[4] Decorator functions are called once for each function they are applied to. Then, each time that function is called in the code, the `wrapper` function is what's actually executed. So imagine applying it to a function like this:

```
@running_average
def foo(x):
    return x + 2
```

This creates an internal dictionary, named `data`, used to keep track of `foo`'s metrics. Running `foo` several times produces:

```
>>> foo(1)
Average of foo so far: 3.00
3
>>> foo(10)
Average of foo so far: 7.50
12
>>> foo(1)
Average of foo so far: 6.00
3
>>> foo(1)
Average of foo so far: 5.25
3
```

The placement of `data` is important. Pop quiz:

[4] In a real application, you'd write the average to some kind of log sink, but we'll use `print()` here because it's convenient for learning.

- What happens if you move the line defining `data` up one line, outside the `running_average` function?
- What happens if you that line down, into the `wrapper` function?

Looking at the code above, decide on your answers to these questions before reading further.

Here's what it looks like if you create `data` outside the decorator:

```
# This version has a bug.
data = {"total" : 0, "count" : 0}
def outside_data_running_average(func):
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

If we do this, *every* decorated function shares the exact same `data` dictionary! This actually doesn't matter if you only ever decorate just one function. But you never bother to write a decorator unless it's going to be applied to at least two:

```
@outside_data_running_average
def foo(x):
    return x + 2

@outside_data_running_average
def bar(x):
    return 3 * x
```

And that produces a problem:

```
>>> # First call to foo...
... foo(1)
Average of foo so far: 3.0
3
>>> # First call to bar...
... bar(10)
Average of bar so far: 16.5
30
>>> # Second foo should still average 3.00!
... foo(1)
Average of foo so far: 12.0
```

Because `outside_data_running_average` uses the *same* `data` dictionary for all the functions it decorates, the statistics are conflated.

Now, the other situation: what if you define `data` inside `wrapper`?

```
# This version has a DIFFERENT bug.
def running_average_data_in_wrapper(func):
    def wrapper(*args, **kwargs):
        data = {"total" : 0, "count" : 0}
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper

@running_average_data_in_wrapper
def foo(x):
    return x + 2
```

Look at the average as we call this decorated function multiple times:

```
>>> foo(1)
Average of foo so far: 3.0
3
>>> foo(5)
Average of foo so far: 7.0
7
>>> foo(20)
Average of foo so far: 22.0
22
```

Do you see why the running average is wrong? The `data` dictionary is reset *every time the decorated function is called*. This is why it's important to consider the scope when implementing your decorator. Here's the correct version again (repeated so you don't have to skip back):

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

So when exactly is `running_average` executed? The decorator function itself is executed **exactly once** for **every** function it decorates. If you decorate N functions, `running_average` is executed N times, so we get N different `data` dictionaries, each tied to one of the resulting decorated functions. This has nothing to do with how many times a decorated function is executed. The decorated function is, basically, one of the created `wrapper` functions. That `wrapper` can be executed many times, using the same `data` dictionary that was in scope when that `wrapper` was defined.

This is why `running_average` produces the correct behavior:

```
@running_average
def foo(x):
    return x + 2

@running_average
def bar(x):
    return 3 * x
```

```
>>> # First call to foo...
... foo(1)
Average of foo so far: 3.0
3
>>> # First call to bar...
... bar(10)
Average of bar so far: 30.0
30
>>> # Second foo gives correct average this time!
... foo(1)
Average of foo so far: 3.0
3
```

Now, what if you want to peek into `data`? The way we've written `running_average`, you can't. `data` persists because of the reference inside of `wrapper`, but there is no way you can access it directly in normal Python code. But when you *do* need to do this, there is a very easy solution: simply assign `data` as an attribute to `wrapper`. For example:

```
# collectstats is much like running_average, but lets
# you access the data dictionary directly, instead
# of printing it out.
def collectstats(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        return func(*args, **kwargs)
    wrapper.data = data
    return wrapper
```

See that line `wrapper.data = data`? Yes, you can do that. A function in Python is just an object, and in Python, you can add new attributes to objects by just assigning them. This conveniently annotates the decorated function:

```
@collectstats
def foo(x):
    return x + 2
```

```
>>> foo.data
{'total': 0, 'count': 0}
>>> foo(1)
3
>>> foo.data
{'total': 3, 'count': 1}
>>> foo(2)
4
>>> foo.data
{'total': 7, 'count': 2}
```

It's clear now why `collectstats` doesn't contain any print statement: you don't need one! We can check the accumulated numbers at any time, because this decorator annotates the function itself, with that `data` attribute.

Let's switch to a another problem you might run into, and how you deal with it. Here's an decorator that counts how many times a function has been called:

```
# Watch out, this has a bug...
count = 0
def countcalls(func):
    def wrapper(*args, **kwargs):
        global count
        count += 1
        print('# of calls: {}'.format(count))
        return func(*args, **kwargs)
    return wrapper

@countcalls
def foo(x): return x + 2

@countcalls
def bar(x): return 3 * x
```

This version of `countcalls` has a bug. Do you see it?

That's right: it stores `count` as a global variable, meaning every function that is decorated will use that same variable:

```
>>> foo(1)
# of calls: 1
3
>>> foo(2)
# of calls: 2
4
>>> bar(3)
# of calls: 3
9
>>> bar(4)
# of calls: 4
12
>>> foo(5)
# of calls: 5
7
```

The solution is trickier than it seems. Here's one attempt:

```
# Move count inside countcalls, and remove the
# "global count" line. But it still has a bug...
def countcalls(func):
    count = 0
    def wrapper(*args, **kwargs):
        count += 1
        print('# of calls: {}'.format(count))
        return func(*args, **kwargs)
    return wrapper
```

But that just creates a different problem:

```
>>> foo(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in wrapper
UnboundLocalError: local variable 'count' referenced before
   assignment
```

We can't use `global`, because it's not global. But in Python 3, we can use the `nonlocal` keyword:

```
# Final working version!
def countcalls(func):
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        print('# of calls: {}'.format(count))
        return func(*args, **kwargs)
    return wrapper
```

This finally works correctly:

```
>>> foo(1)
# of calls: 1
3
>>> foo(2)
# of calls: 2
4
>>> bar(3)
# of calls: 1
9
>>> bar(4)
# of calls: 2
12
>>> foo(5)
# of calls: 3
```

Applying `nonlocal` gives the `count` variable a special scope that is part-way between local and global. Essentially, Python will search for the nearest enclosing scope that defines a variable named `count`, and use it like it's a global.[5]

You may be wondering why we didn't need to use `nonlocal` with the first version of `running_average` above - here it is again, for reference:

[5] `nonlocal` is not available in Python 2; if you are using that version, see the next section.

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
        data["count"] += 1
        print("Average of {} so far: {:.01f}".format(
             func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

When we have a line like `count += 1`, that's actually modifying the value of the `count` variable itself - because it really means `count = count + 1`. And whenever you modify (instead of just read) a variable that was created in a larger scope, Python requires you to declare that's what you actually want, with `global` or `nonlocal`.

Here's the sneaky thing: when we write `data["count"] += 1`, **that is not actually modifying `data`!** Or rather, it's not modifying the *variable* named `data`, which points to a dictionary object. Instead, the statement `data["count"] += 1` invokes a *method* on the `data` object. This does change the state of the dictionary, but it doesn't make `data` point to a *different* dictionary. But `count +=1` makes `count` point to a different integer, so we need to use `nonlocal` there.

2.2.1 Data in Decorators for Python 2

The `nonlocal` keyword didn't exist before version 3.0, so Python 2 has no way to say "this variable is partway between local and global". But you have several workarounds.

My favorite technique is to assign the variable as an attribute to the `wrapper` function:

```
def countcalls(func):
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        print('# of calls: {}'.format(wrapper.count))
        return func(*args, **kwargs)
    wrapper.count = 0
    return wrapper
```

Instead of a variable named `count`, the `wrapper` function object gets an attribute named `count`. So everywhere inside `wrapper`, I reference the variable as `wrapper.count`. One interesting thing is that this `wrapper.count` variable is initialized *after* the function is defined, just before the final `return` line in `countcalls`. Python has no problem with this; the attribute doesn't exist when `wrapper` is defined, but so long as it exists when the decorated function is *called* for the first time, no error will result.

This is my favorite solution, and what I use in my own Python 2 code. It's not commonly used, however, so I will explain a couple of other techniques you may see. One is to use `hasattr` to check whether `wrapper.count` exists yet, and if not, initialize it:

```python
def countcalls(func):
    def wrapper(*args, **kwargs):
        if not hasattr(wrapper, 'count'):
            wrapper.count = 0
        wrapper.count += 1
        print('# of calls: {}'.format(wrapper.count))
        return func(*args, **kwargs)
    return wrapper
```

This has a slight performance disadvantage, because `hasattr` will be called every time the decorated function is invoked, while the first approach does not. It's unlikely to matter unless you're deeply inside some nested for-loop, though.

Alternatively - and this one has no performance disadvantage - you can create a `list` object just outside of `wrapper`'s scope, and treat its first element like the variable you want to change:

```python
def countcalls(func):
    count_container = [0]
    def wrapper(*args, **kwargs):
        print('# of calls: {}'.format(count_container[0]))
        count_container[0] += 1
        return func(*args, **kwargs)
    return wrapper

@countcalls
def foo(x): return x + 2

@countcalls
def bar(x): return 3 * x
```

Basically, everywhere the Python 3 version would say `count`, your code will say `count_container[0]`. This works without needing `global` or `nonlocal`, because the `count_container` *contents* are modified, but it doesn't modify what the `count_container` *variable* points to. In other words, it's always the same list; you're just changing the first (and only) element in that list. A bit clunky, but probably closest in spirit to what you can do in Python 3 with nonlocal.

2.3 Decorators That Take Arguments

Early in the chapter I showed you an example decorator from the Flask framework:

```
@app.route("/")
def hello():
    return "<html><body>Hello World!</body></html>"
```

This is different from any decorator we've implemented so far, because it actually takes an argument. How do we write decorators that can do this? For example, imagine a family of decorators adding a number to the return value of a function:

```
def add2(func):
    def wrapper(n):
        return func(n) + 2
    return wrapper

def add4(func):
    def wrapper(n):
        return func(n) + 4
    return wrapper

@add2
def foo(x):
    return x ** 2

@add4
def bar(n):
    return n * 2
```

There is literally only one character difference between add2 and add4; it's very repetitive, and poorly maintainable. Wouldn't it be better if we can do something like this:

```python
@add(2)
def foo(x):
    return x ** 2

@add(4)
def bar(n):
    return n * 2
```

We can. The key is to understand that add is actually *not* a decorator; it is a function that *returns* a decorator. In other words, add is a function that returns another function. (Since the returned decorator is, itself, a function).

To make this work, we write a function called add, which creates and returns the decorator:

```python
def add(increment):
    def decorator(func):
        def wrapper(n):
            return func(n) + increment
        return wrapper
    return decorator
```

It's easiest to understand from the inside out:

- The `wrapper` function is just like in the other decorators. Ultimately, when you call `foo` (the original function name), it's actually calling `wrapper`.
- Moving up, we have the aptly named `decorator`. Hint: we could say `add2 = add(2)`, then apply `add2` as a decorator.
- At the top level is `add`. This is not a decorator. It's a function that returns a decorator.

Notice the closure here. The `increment` variable is encapsulated in the scope of the add function. We can't access its value outside the decorator, in the calling context. But we don't need to, because `wrapper` itself has access to it.

Suppose the Python interpreter is parsing your program, and encounters the following code:

```python
@add(2)
def f(n):
    # ....
```

Python takes everything between the @-symbol and the end-of-line character as a single Python expression - that would be “add(2)” in this case. That expression is evaluated. This all happens *at compile time*. Evaluating the decorator expression means executing `add(2)`, which will return a function object. That function object is the decorator. It’s named `decorator` inside the body of the add function, but it doesn’t really have a name at the top level; it’s just applied to `f`.

What can help you see more clearly is to think of functions as things that are stored in variables. In other words, if I write `def foo(x):`, in my code, I could say to myself "I’m creating a function called foo". But there is another way to think about it. I can say "I’m creating a function object, and storing it in a variable called foo". Believe it or not, this is actually much closer to how Python actually works. So things like this are possible:

```
>>> def foo():
...     print("This is foo")
>>> baz = foo
>>> baz()
This is foo
>>> # foo and baz have the same id()... so they
... # refer to the same function object.
>>> id(foo)
4301663768
>>> id(baz)
4301663768
```

Now, back to add. As you realize `add(2)` returns a function object, it’s easy to imagine storing that in a variable named `add2`. As a matter of fact, the following are all exactly equivalent:

```
# This...
add2 = add(2)
@add2
def foo(x):
    return x ** 2

# ... has the same effect as this:
@add(2)
def foo(x):
    return x ** 2
```

Remember that `@` is a shorthand:

```python
# This...
@some_decorator
def some_function(arg):
    # blah blah

# ... is translated by Python into this:
def some_function(arg):
    # blah blah
some_function = some_decorator(some_function)
```

So for add, the following are all equivalent:

```python
add2 = add(2) # Store the decorator in the add2 variable

# This function definition...
@add2
def foo(x):
    return x ** 2

# ... is translated by Python into this:
def foo(x):
    return x ** 2
foo = add2(foo)

# But also, this...
@add(2)
def foo(x):
    return x ** 2

# ... is translated by Python into this:
def foo(x):
    return x ** 2
foo = add(2)(foo)
```

Look over these four variations, and trace through what's going on in your mind, until you understand how they are all equivalent. The expression `add(2)(foo)` in particular is interesting. Python parses this left-to-right. So it first executes `add(2)`, which returns a function object. In this expression, that function has no name; it's temporary and anonymous. Python takes that anonymous function object, and immediately calls it, with the argument `foo`. (That argument is, of course, the bare function - the function which we are decorating, in other words.)

The anonymous function then returns a *different* function object, which we finally store in the variable called `foo`.

Notice that in the line `foo = add(2)(foo)`, the name `foo` means something different each time it's used. Just like when you write something like `n = n + 3`; the name `n` refers to something different on either side of the equals sign. In the exact same way, in the line `foo = add(2)(foo)`, the variable `foo` holds two different function objects on the left and right sides.

2.4 Class-based Decorators

I lied to you.

I repeatedly told you a decorator is just a function. Well, decorators are usually *implemented* as functions, that's true. However, it's also possible to implement a decorator using classes. In fact, *any* decorator that you can implement as a function can be done with a class instead.

Why would you do this? Basically, for certain kinds of more complex decorators, classes are better suited, more readable, or otherwise easier to work with. For example, if you have a collection of related decorators, you can leverage inheritance or other object-oriented features. Simpler decorators are better implemented as functions, though it depends on your preferences for OO versus functional abstractions. It's best to learn both ways, then decide which you prefer in your own code on a case-by-case basis.

The secret to decorating with classes is the magic method `__call__`. Any object can implement `__call__` to make it callable - meaning, the object can be called like a function. Here's a simple example:

```
class Prefixer:
    def __init__(self, prefix):
        self.prefix = prefix
    def __call__(self, message):
        return self.prefix + message
```

You can then, in effect, "instantiate" functions:

```
>>> simonsays = Prefixer("Simon says: ")
>>> simonsays("Get up and dance!")
'Simon says: Get up and dance!'
```

Just looking at `simonsays("Get up and dance!")` in isolation, you'd never guess it is anything other than a normal function. In fact, it's an instance of `Prefixer`.

You can use `__call__` to implement decorators, in a very different way. Before proceeding, quiz yourself: thinking back to the `@printlog` decorator, and using this information about `__call__`, how might you implement `printlog` as a class instead of a function?

The basic approach is to pass `func` it to the *constructor* of a decorator *class*, and adapt `wrapper` to be the `__call__` method:

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)

# Compare to the function version you saw earlier:
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

```
>>> @PrintLog
... def foo(x):
...     print(x + 2)
...
>>> @PrintLog
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

From the point of view of the user, `@Printlog` and `@printlog` work *exactly* the same.

Class-based decorators have a few advantages over function-based. For one thing, the decorator is a class, which means you can leverage inheritance. So if you have a family of related decorators, you can reuse code between them. Here's an example:

```
import sys
class ResultAnnouncer:
    stream = sys.stdout
    prefix = "RESULT"
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        value = self.func(*args, **kwargs)
        self.stream.write('{}: {}\n'.format(self.prefix, value))
        return value

class StdErrResultAnnouncer(ResultAnnouncer):
    stream = sys.stderr
    prefix = "ERROR"
```

Another benefit is when you prefer to accumulate state in object attributes, instead of a closure. For example, the `countcalls` function decorator above could be implemented as a class:

```
class CountCalls:
    def __init__(self, func):
        self.func = func
        self.count = 1
    def __call__(self, *args, **kwargs):
        print('# of calls: {}'.format(self.count))
        self.count += 1
        return self.func(*args, **kwargs)

@CountCalls
def foo(x):
    return x + 2
```

Notice this also lets us access `foo.count`, if we want to check the count outside of the decorated function. The function version didn't let us do this.

When creating decorators which take arguments, the structure is a little different. In this case, the constructor accepts not the `func` object to be decorated, but the parameters on the decorator line. The `__call__` method must take the `func` object, define a wrapper function, and return it - similar to simple function-based decorators:

```
# Class-based version of the "add" decorator above.
class Add:
    def __init__(self, increment):
        self.increment = increment
    def __call__(self, func):
        def wrapper(n):
            return func(n) + self.increment
        return wrapper
```

You then use it in a similar manner to any other argument-taking decorator:

```
>>> @Add(2)
... def foo(x):
...     return x ** 2
...
>>> @Add(4)
... def bar(n):
...     return n * 2
...
>>> foo(3)
11
>>> bar(77)
158
```

Any function-based decorator can be implemented as a class-based decorator; you simply adapt the decorator function itself to `__init__`, and `wrapper` to `__call__`. It's possible to design class-based decorators which cannot be translated into a function-based form, though.

For complex decorators, some people feel that class-based are easier to read than function-based. In particular, many people seem to find multiply nested `def`'s hard to reason about. Others (including your author) feel the opposite. This is a matter of preference, and I recommend you practice with both styles before coming to your own conclusions.

2.5 Decorators For Classes

I lied to you again. I said decorators are applied to functions and methods. Well, they can also be applied to classes.

(Understand this has *nothing* to do with the last section's topic, on implementing decorators as classes. A decorator can be implemented as a function, or as a class; and that decorator can be applied to a function, or to a class. They are independent ideas; here, we are talking about how to decorate classes instead of functions.)

To introduce an example, let me explain Python's built-in `repr()` function. When called with one argument, this returns a string, meant to represent the passed object. It's similar to `str()`; the difference is that while `str()` returns a human-readable string, `repr()` is meant to return a string version of the Python code needed to recreate it. So imagine a simple `Penny` class:

```
class Penny:
    value = 1

penny = Penny()
```

Ideally, `repr(penny)` returns the string `"Penny()"`. But that's not what happens by default:

```
>>> class Penny:
...     value = 1
>>> penny = Penny()
>>> repr(penny)
'<__main__.Penny object at 0x10229ff60>'
```

You fix this by implementing a `__repr__` method on your classes, which `repr()` will use:

```
>>> class Penny:
...     value = 1
...     def __repr__(self):
...         return "Penny()"
>>> penny = Penny()
>>> repr(penny)
'Penny()'
```

You can create a decorator that will automatically add a `__repr__` method to any class. You might be able to guess how it works. Instead of a wrapper function, the decorator returns a class:

```
>>> def autorepr(klass):
...     def klass_repr(self):
...         return '{}()'.format(klass.__name__)
...     klass.__repr__ = klass_repr
...     return klass
...
>>> @autorepr
... class Penny:
...     value = 1
...
>>> penny = Penny()
>>> repr(penny)
'Penny()'
```

It's suitable for classes with no-argument constructors, like `Penny`. Note how the decorator modifies `klass` directly. The original class is returned; that original class just now has a `__repr__` method. Can you see how this is different from what we did with decorators of functions? With those, the decorator returned a new, different function object.

Another strategy for decorating classes is closer in spirit: creating a new subclass within the decorator, returning that in its place:

```
def autorepr_subclass(klass):
    class NewClass(klass):
        def __repr__(self):
            return '{}()'.format(klass.__name__)
    return NewClass
```

This has the disadvantage of creating a new type:

```
>>> @autorepr_subclass
... class Nickel:
...     value = 5
...
>>> nickel = Nickel()
>>> type(nickel)
<class '__main__.autorepr_subclass.<locals>.NewClass'>
```

The resulting object's type isn't obviously related to the decorated class. That makes debugging harder, creates unclear log messages, and has other unexpected effects. For this reason, I recommend you prefer the first approach.

Class decorators tend to be less useful in practice than those for functions and methods. When they are used, it's often to automatically generate and add methods. But they are more flexible than that. You can even express the singleton pattern using class decorators:

```
def singleton(klass):
     instances = {}
     def get_instance():
         if klass not in instances:
             instances[klass] = klass()
         return instances[klass]
     return get_instance

# There is only one Elvis.
@singleton
class Elvis:
    pass
```

Note the IDs are the same:

```
>>> elvis1 = Elvis()
>>> elvis2 = Elvis()
>>>
>>> id(elvis1)
4333747560
>>> id(elvis2)
4333747560
```

2.6 Preserving the Wrapped Function

The techniques in this chapter for creating decorators are time-tested, and valuable in many situations. But the resulting decorators have a few problems:

- Function objects automatically have certain attributes, like `__name__`, `__doc__`, `__module__`, etc. The wrapper clobbers all these, breaking any code relying on them.
- Decorators interfere with introspection - masking the wrapped function's signature, and blocking `inspect.getsource()`.
- Decorators cannot be applied in certain more exotic situations - like class methods, or descriptors - without going through some heroic contortions.

The first problem is easily solved using the standard library's `functools` module. It includes a function called `wraps`, which you use like this:

```
import functools
def printlog(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print('CALLING: {}'.format(func.__name__))
        return func(*args, **kwargs)
    return wrapper
```

That's right - `functools.wraps` is a decorator, that you use *inside* your own decorator. When applied to the `wrapper` function, it essentially copies certain attributes from the wrapped function to the wrapper. It is equivalent to this:

```
def printlog(func):
    def wrapper(*args, **kwargs):
        print('CALLING: {}'.format(func.__name__))
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    wrapper.__module__ = func.__module__
    wrapper.__annotations__ = func.__annotations__
    return wrapper
```

```
>>> @printlog
... def foo(x):
...     "Double-increment and print number."
...     print(x + 2)
...
>>> # functools.wraps transfers the wrapped function's attributes
... foo.__name__
'foo'
>>> print(foo.__doc__)
Double-increment and print number.
```

Contrast this with the default behavior:

```
# What you get without functools.wraps.
>>> foo.__name__
'wrapper'
>>> print(foo.__doc__)
None
```

In addition to saving you lots of tedious typing, `functools.wraps` encapsulates the details of *what* to copy over, so you don't need to worry if new attributes are introduced in future versions of Python. For example, the `__annotations__` attribute was added in Python 3; those who used `functools.wraps` in their Python 2 code had one less thing to worry about when porting to Python 3.

`functools.wraps` is a actually a convenient shortcut of the more general `update_wrapper`. Since `wraps` only works with function-based decorators, your class-based decorators must use `update_wrapper` instead:

```
import functools
class PrintLog:
    def __init__(self, func):
        self.func = func
        functools.update_wrapper(self, func)
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)
```

While useful for copying over `__name__`, `__doc__`, and the other attributes, `wraps` and `update_wrapper` do not help with the other problems mentioned above. The closest to a full solution is Graham Dumpleton's `wrapt` library.[6] Decorators created using the `wrapt` module work in situations that cause normal decorators to break, and behave correctly when used with more exotic Python language features.

So what should you do in practice?

Common advice says to proactively use `functools.wraps` in all your decorators. I have a different, probably controversial opinion, born from observing that most Pythonistas in the wild do *not* regularly use it, including myself, even though we know the implications.

While it's true that using `functools.wraps` on all your decorators will prevent certain problems, doing so is not completely free. There is a cognitive cost, in that you have to remember

[6] `pip install wrapt`. See also https://github.com/GrahamDumpleton/wrapt and http://wrapt.readthedocs.org/ .

to use it - at least, unless you make it an ingrained, fully automatic habit. It's boilerplate which takes extra time to write, and which references the `func` parameter - so there's something else to modify if you change its name. And with `wrapt`, you have another library dependency to manage.

All these amount to a small distraction each time you write a decorator. And when you *do* have a problem that `functools.wraps` or the `wrapt` module would solve, you are likely to encounter it during development, rather than have it show up unexpectedly in production. (Look at the list above again, and this will be evident.) When that happens, you can just add it and move on.

The biggest exception is probably when you are using some kind of automated API documentation tool,[7] which will use each function's `__doc__` attribute to generate reference docs. Since decorators systematically clobber that attribute, it makes sense to document a policy of using `functools.wraps` for all decorators in your coding style guidelines, and enforce it in code reviews.

Aside from situations like this, though, the problems with decorators will be largely theoretical for most (but not all) developers. If you are in that category, I recommend optimistically writing decorators *without* bothering to use `wraps`, `update_wrapper`, or the `wrapt` module. If and when you realize you are having a problem that these would solve for a specific decorator, introduce them then.[8]

[7] See https://wiki.python.org/moin/DocumentationTools for a thorough list.

[8] A perfect example of this happens towards the end of the "Building a RESTful API Server in Python" video (https://powerfulpython.com/store/restful-api-server/), when I create the `validate_summary` decorator. Applying the decorator to a couple of Flask views immediately triggers a routing error, which I then fix using `wraps`.

Index