Python: The Next Level

Aaron Maxwell

aaron@powerfulpython.com

Contents

1 Advanced Functions

In this chapter, we go beyond the basics of using functions. I'll assume you can fluently define and use 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
```

The topics covered in this chapter are not only useful and valuable on their own. They are also important building blocks for some extremely powerful Python programming idioms, 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:

```
def takes_any_args(*args):
    message = "Got args: "
    for arg in args:
        message += str(arg) + " "
    print(message)
```

Note 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")
Got args: x y z
>>> takes_any_args(1)
Got args: 1
>>> takes_any_args()
Got args:
>>> takes_any_args(5, 4, 3, 2, 1)
Got args: 5 4 3 2 1
>>> takes_any_args(["first", "list"], ["another","list"])
Got 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("items: {}".format(items))
...
>>> takes_a_list(["x", "y", "z"])
items: ['x', 'y', 'z']
>>> takes_any_args(["x", "y", "z"])
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 any valid 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` make the code more understandable, then do it.

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 unpack the tuple from `get_required_textbook` manually, and then pass it to `order_book`:

```
>>> title, author, isbn = get_required_textbook(4242)
>>> order_book(title, author, isbn)
Ordering 'Writing Great Code' by Randall Hyde (1593270038)
```

[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.

Alternatively you could do this, which is pretty horrible:

```
>>> 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)
```

Not ideal; writing code this way is tedious and error-prone. Fortunately, Python lets us pass a tuple of arguments to a normal function, unpacking them in place. All we have to do is put an asterisk before the argument in the function call:

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

Notice `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 kind of duality here. We can use the asterisk syntax both in *defining* a function, and in *calling* a function. The syntax looks very similar. But that is slightly misleading; they are doing two different, yet related things. One is packing arguments into a tuple automatically, the other is *un*-packing them. It's helpful to be very clear on the distinction between the two in your mind.

Armed with this complete understanding, we can bridge these 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 normal, *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....
>>> get_rental_cars("midsize", doors=2)
Looking for a 2-door midsize car with automatic transmission....
```

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

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

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_inverted(hero="Homer", antihero="Bart", genius="Lisa")
Bart <- antihero
Homer <- hero
Lisa <- genius
```

The arguments to `print_inverted` are key-value pairs. This is very normal Python syntax for calling functions; what's interesting is happening *inside* the function. There, a variable called `kwargs` is defined. It's a boringly normal 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:

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

```
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:

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

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 technically called *keyword argument unpacking*, but people often just say "using kwargs". Again, naming this variable kwargs is just a strong convention; you can choose a different name if that improves readability.

Using kwargs 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 literally be called in any way, with any 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 1, 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 before "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 Grimwald'}
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:

[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. In practice, key functions are frequently actual functions, but can be any callable.

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

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, just like we created above:

```
>>> 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 Grimwald'}
>>> 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 Grimwald'},
 {'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 Grimwald'},
 {'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 Grimwald'},
 {'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 Grimwald", "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 Grimwald', '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 Grimwald", "literature", 3.4),
...      ]
>>> from operator import attrgetter
>>> sorted(student_objs, key=attrgetter("gpa"))
[Zoe Grimwald: 3.4, Joe Smith: 3.7, Jane Jones: 3.8]
```

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.[6]

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`.[7] 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)
```

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:

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

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

- 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.[8]

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:

```
# 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:

[8]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.

```
>>> 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.

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.[9] 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`, that is 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:

[9] 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 create data outside the running_average function, at the top level?
- What happens if you create data inside the wrapper function?

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

Here's what it looks like it 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 in real code, you almost 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:

```
# Correct version - repeated here 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.[10]

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:

```
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**

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

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 or global 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 also see. One is to use hasattr to check whether wrapper.count exists yet, and if not, initialize it:

```
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:

```
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 web frameworkFlask 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:

```
@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:

```
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:

```
@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()... which means these
... # variables are storing the same object, the same function.
... id(foo)
4310262784
>>> id(baz)
4310262784
```

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:

```
# 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:

```
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.

2.3.1 Extra Credit: A Webapp Framework

Lab file: `labs/decorators/webframework.py`.

The Flask web framework allows you to register handler functions to URLs using a decorator-based syntax:

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

Using what you now know about decorators, you will implement a class called `WebApp` in this lab that lets you do something similar. Create the class from scratch, giving it the methods and state it needs to make all the tests pass.

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):
    prefix = "ERROR: "
    stream = sys.stderr
```

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.)

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()'
```

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 saw above 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 implement a simple 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.[11] 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.

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

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 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 is probably when you are using some kind of automated API documentation tool[12], 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.[13]

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

[13] A perfect example of this happens towards the end of the "Building a RESTful API Server in Python" video, 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`.

3 Classes and Objects: Beyond The Basics

This chapter assumes you are familiar with Python's OOP basics: creating classes, defining methods, and using inheritance. We build on this.

As with any object-oriented language, it's useful to learn about **design patterns** - reusable solutions to common problems involving classes and objects. A LOT has been written about design patterns. Curiously, though, much of what's out there doesn't completely apply to Python - or, at least, it applies *differently*.

That's because many of these design-pattern books, articles, and blog posts are for languages like Java, C++ and C#. But as a language, Python is quite different. Its dynamic typing, first-class functions, and other additions all mean the "standard" design patterns just work differently.

So let's learn what Pythonic OOP is *really* about.

3.1 Quick Note on Python 2

This chapter uses Python 3 syntax. Python 2.7 is nearly the same, and I'll point out the minor differences as we go along. But there is one *big* difference worth emphasizing here.

In modern Python, all classes need to inherit from a built-in base class called `object`. (It's lowercased, defying the normal convention.) This happens automatically for all classes in Python 3:

```
>>> # Python 3
... class Dog:
...     def speak(self):
...         return "woof"
...
>>> dog = Dog()
>>> isinstance(dog, object)
True
```

In Python 2, you must explicitly inherit your classes from `object`. Fail to do this, and your class builds on "old-style classes":

```
>>> # Python 2
... class DogFromObject(object):
...     def speak(self):
...         return "woof"
...
>>> class DogNotFromObject:
...     def speak(self):
...         return "woof"
...
>>> issubclass(DogFromObject, object)
True
>>> issubclass(DogNotFromObject, object)
False
```

If you don't already base your Python 2 classes on `object`, start today. Old-style classes are long obsolete, and removed in Python 3; they partially or completely break many important tools in Python's object system, like properties and `super()`. The rest of this chapter assumes you're inheriting from `object`.

3.2 Properties

In object-oriented programming, a *property* is a special sort of class member. It's almost a cross between a method and an attribute. The idea is that you can, when designing the class, create "attributes" whose reading, writing, and so on can be managed by special methods. In Python, you do this with a decorator named `property`. Here's an example:

```
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
```

By instantiating this, I can access `full_name` as a kind of virtual attribute:

```
>>> joe = Person("Joe", "Smith")
>>> joe.full_name
'Joe Smith'
```

Notice carefully the members here: there are two attributes called `first_name` and `last_name`, set in the constructor. There is also a method called `full_name`. But after creating the object, we reference `joe.full_name` as an attribute; we don't call `joe.full_name()` as a method.

This is all due to the `@property` decorator. When applied to a method, this decorator makes it inaccessible as a method. You must access it as an attribute. In fact, if you try to call it as a method, you get an error:

```
>>> joe.full_name()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

As defined above, `full_name` is read-only. We can't modify it:

```
>>> joe.full_name = "Joseph Smith"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In other words, Python properties are read-only by default. Another way of saying this is that `@property` automatically defines a *getter*, but not a *setter*. If you do want to `full_name` to be writable, here is how you define the setter:

```
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

    @full_name.setter
    def full_name(self, value):
        self.first_name, self.last_name = value.split(" ", 1)
```

This lets us assign to `joe.full_name`:

```
>>> joe = Person("Joe", "Smith")
>>> joe.first_name
'Joe'
>>> joe.last_name
'Smith'
>>> joe.full_name = "Joseph Smith"
>>> joe.first_name
'Joseph'
>>> joe.last_name
'Smith'
```

The first time I saw this, I had all sorts of questions. "Wait, why is `full_name` defined twice? And why is the second decorator named `@full_name`, and what's this `setter` attribute? How on earth does this even byte compile?"

The code is actually correct, and designed to work this way. The `@property` `def` `full_name` must come first. That creates the property to begin with, and *also* creates the getter. By "create the property", I mean that an object named `full_name` exists *in the namespace of the class*, and it has a method named `full_name.setter`. This `full_name.setter` is a decorator that is applied to the next `def full_name`, christening it as the setter for the `full_name` property.

It's okay to not fully understand how this all works. A full explanation relies on understanding both implementing decorators, and Python's descriptor protocol, both of which are beyond the scope of what we want to focus on here. Fortunately, you don't have to understand *how* it works in order to use it.

Besides getting and setting, properties have a couple of other options which are less commonly used. `full_name.deleter` can be used as a decorator to handle the `del` operation for the object attribute. This seems to be rarely needed in practice, but it's available when you do.

What you see here with the `Person` class is one way properties are useful: magic attributes whose values are derived from other values. This denormalizes the object's data, and lets you access the property value as an attribute instead of as a method. (The benefit of this is sometimes more cognitive than anything else.)

Properties enable a useful collection of design patterns. One - as mentioned - is in creating read-only member variables. In `Person`, the `full_name` "member variable" is a dynamic attribute; it doesn't exist on its own, but instead calculates its value at run-time.

It’s also common to have the property backed by a single, non-public member variable. That pattern looks like this:

```python
class Coupon:
    def __init__(self, amount):
        self._amount = amount
    @property
    def amount(self):
        return self._amount
```

This allows the class itself to modify the value internally, but prevent outside code from doing so:

```
>>> coupon = Coupon(1.25)
>>> coupon.amount
1.25
>>> coupon.amount = 1.50
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In Python, prefixing a member variable by a single underscore signals the variable is non-public, i.e. it should only be accessed internally, inside methods of that class, or its subclasses.[14] What this pattern says is "you can access this variable, but not change it".

Between "regular member variable" and "ready-only" is another pattern: allow changing the attribute, but validate it first. Let me explain. Suppose my event-management application has a `Ticket` class, representing tickets sold to concert-goers:

```python
class Ticket:
    def __init__(self, price):
        self.price = price
    # And some other methods...
```

One day, we find a bug in our web UI, which lets some shifty customers adjust the price to a negative value... so we ended up actually *paying* them to go to the concert. Not good!

The first priority is, of course, to fix the bug in the UI. But do we modify our code to prevent this from ever happening again? Before reading further, look at the `Ticket` class and ponder - how could you use properties to make this kind of bug impossible in the future?

[14]This isn’t enforced by Python itself. If your teammates don’t already honor this widely-followed convention, you’ll have to educate them.

The answer: verify the new price is non-zero in the setter:

```python
# Version 1...
class Ticket:
    def __init__(self, price):
        self._price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

This lets the price be adjusted... but only to sensible values:

```
>>> t = Ticket(42)
>>> t.price = 24 # This is allowed.
>>> print(t.price)
24
>>> t.price = -1 # This is NOT.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in price
ValueError: Nice try
```

However, there's a defect in this new `Ticket` class. Can you spot what it is? (And how to fix it?)

The problem is that while we can't *change* the price to a negative value, this first version lets us *create* a ticket with a negative price to begin with. That's because we write `self._price = price` in the constructor. The solution is to use the *setter* in the constructor instead:

```
# Final version, with modified constructor.
# (Constructor is different; code for getter & setter is the same ↩
   .)
class Ticket:
    def __init__(self, price):
        # instead of "self._price = price"
        self.price = price
    @property
    def price(self):
        return self._price
    @price.setter
    def price(self, new_price):
        # Only allow positive prices.
        if new_price < 0:
            raise ValueError("Nice try")
        self._price = new_price
```

Yes, you can reference `self.price` in methods of the class. When we write `self.price = price`, Python translates this to calling the `price` setter - i.e., the second `price()` method. This final version of `Ticket` centralizes all reads AND writes of `self._price` in the property. It's a useful encapsulation principle in general. The idea is you centralize any special behavior for that member variable in the getter and setter, even for the class's internal code. In practice, sometimes you will find you need to violate this rule; you simply reference `self._price` and move on. But if you avoid that as long as you can, you will probably benefit from higher quality code.

3.2.1 Properties and Refactoring

Properties are important in most languages today. Here's a situation that often plays out. Imagine writing a simple money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    # And some other methods...
```

Suppose you put this class in a library, which many developers are using. People on your current team, perhaps developers on different teams. Or maybe you release it as open-source, so developers around the world use and rely on this class.

Now, one day you realize many of Money's methods - which do calculations on the money amount - can be simpler and more straightforward if they operate on the total number of cents, rather than dollars and cents separately. So you refactor the internal state:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
```

This minor change creates a MAJOR maintainability problem. Can you spot it?

Here's the trouble: your original Money has attributes named dollars and cents. And since many developers are using these, changing to total_cents breaks all their code!

```
money = Money(27, 12)
message = "I have {:d} dollars and {:d} cents."
# This line breaks, because there's no longer
# dollars or cents attributes.
print(message.format(money.dollars, money.cents))
```

If no one but you uses this class, there's no real problem - you can just refactor your own code. But if that's not the case, coordinating this change with everyone's different code bases is a nightmare. It becomes a barrier to improving your own code.

So, what do you do? Can you think of a way to handle this situation?

You get out of this mess is with properties. You want two things to happen:

1. Use `total_cents` internally, and
2. All code using `dollars` and `cents` continues to work, without modification.

You do this by replacing `dollars` and `cents` with `total_cents` internally, but also creating getters and setters for these attributes. Take a look:

```
class Money:
    def __init__(self, dollars, cents):
        self.total_cents = dollars * 100 + cents
    # Getter and setter for dollars...
    @property
    def dollars(self):
        return self.total_cents // 100
    @dollars.setter
    def dollars(self, new_dollars):
        self.total_cents = 100 * new_dollars + self.cents
    @property
    def cents(self):
        return self.total_cents % 100
    @cents.setter
    def cents(self, new_cents):
        self.total_cents = 100 * self.dollars + new_cents
```

Now, I can get and set `dollars` and `cents` all day:

```
>>> money = Money(27, 12)
>>> money.total_cents
2712
>>> money.cents
12
>>> money.dollars = 35
>>> money.total_cents
3512
```

Python's way of doing properties brings many benefits. In languages like Java, the following story often plays out:

1. A newbie developer starts writing Java classes. They want to expose some state, so create public member variables.
2. They use this class everywhere in their code base. Other developers do too.
3. One day, they want to change the name or type of that member variable, or even do away with it entirely (like what we did with `Money`).
4. But that would break everyone's code. So they can't.

Because of this, Java developers quickly learn to make all their variables private by default - proactively creating getters and setters for *every* publicly exposed chunk of data. They realize this boilerplate is far less painful than the alternative, because if everyone must use the public getters and setters to begin with, you always have the freedom to make internal changes later.

This works well enough. But it *is* distracting, and just enough trouble that there's always the temptation to make that member variable public, and be done with it.

In Python, we have the best of both worlds. We make member variables public by default, refactoring them as properties if and when we ever need to. No one using our code even has to know.

3.3 The Factory Patterns

There are several design patterns with the word "factory" in their names. Their unifying idea is providing a handy, simplified way to create useful, potentially complex objects. There two most important forms are:

- Where the object's type is fixed, but we want to have several different ways to create it. This is called the *Simple Factory Pattern*.
- Where the factory dynamically chooses one of several different types. This is called the *Factory Method Pattern*.

Let's look at how you do these in Python.

3.3.1 Alternative Constructors: The Simple Factory

Imagine a simple `Money` class, suitable for currencies which have dollars and cents:

```python
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
```

We looked at this in the previous section, refactoring its attributes - but let's roll back, and focus instead on the constructor's interface. This constructor is convenient when we have the dollars and cents as separate integer variables. But there are many other ways to specify an amount of money. Perhaps you're modeling a giant jar of pennies:

```
# Emptying the penny jar...
total_pennies = 3274
# // is integer division
dollars = total_pennies // 100
cents = total_pennies % 100
total_cash = Money(dollars, cents)
```

Suppose your code splits pennies into dollars and cents over and over, and you're tired of repeating this calculation. You could change the constructor, but that means refactoring all `Money`-creating code, and perhaps a lot of code fits the current constructor better anyway. Some languages let you define several constructors, but Python makes you pick one.

In this case, you can usefully create a *factory function* taking the arguments you want, creating and returning the object.

```
# Factory function taking a single argument, returning
# an appropriate Money instance.
def money_from_pennies(total_cents):
    dollars = total_cents // 100
    cents = total_cents % 100
    return Money(dollars, cents)
```

Imagine that, in the same code base, you also routinely need to parse a string like `"$140.75"`. Here's another factory function for that:

```
# Another factory function, creating Money from a string amount.
import re
def money_from_string(amount):
    match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', ↩
       amount)
    assert match is not None, 'Invalid amount: {}'.format(amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

These are effectively alternate constructors: callables we can use with different arguments, which are parsed and used to create the final object. But this approach has problems. First, it's awkward to have them as separate functions, defined outside of the class. But much more importantly: what happens if you subclass `Money`? Suddenly `money_from_string` and `money_from_pennies` are worthless. The base `Money` class is hard-coded.

Python solves these problems in unique way, absent from other languages: the `classmethod` decorator. Use it like this:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
```

The function `money_from_pennies` is now a method of the `Money` class, called `from_pennies`. But it has a new argument: `cls`. When applied to a method definition, `classmethod` modifies how that method is invoked and interpreted. The first argument is not `self`, which would be an *instance* of the class. The first argument is now *the class itself*. In the method body, `self` isn't mentioned at all; instead, `cls` is a variable holding the current class object - `Money` in this case. So the last line is creating a new instance of `Money`:

```
>>> piggie_bank_cash = Money.from_pennies(3217)
>>> type(piggie_bank_cash)
<class '__main__.Money'>
>>> piggie_bank_cash.dollars
32
>>> piggie_bank_cash.cents
17
```

Notice `from_pennies` is invoked off the class itself, not an instance of the class. This already is nicer code organization. But the real benefit is with inheritance:

```
>>> class TipMoney(Money):
...     pass
...
>>> tip = TipMoney.from_pennies(475)
>>> type(tip)
<class '__main__.TipMoney'>
```

This is the *real* benefit of class methods. You define it once on the base class, and all subclasses can leverage it, substituting their own type for `cls`. **This makes class methods perfect for the**

simple factory in Python. Here's the complete `Money` class, with

Notice `self` is not mentioned anywhere in the body. The final line returns an instance of `cls`, using its regular constructor. Here, `cls` is `Money`, so that last line is exactly equivalent to the freestanding function version above.

For the record, here's how we translate `money_from_string`:

```
    @classmethod
    def from_string(cls, amount):
        match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$ ↩
           ', amount)
        assert match is not None, 'Invalid amount: {}'.format( ↩
           amount)
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

Class methods are a superior way to implement factories like this in Python. If we subclass `Money`, that subclass will have `from_pennies` and `from_string` methods that create objects of that subclass, without any extra work on our part. And if we change the name of the `Money` class, we only have to change it in one place, not three.

This form of the factory pattern is called "simple factory", a name I don't love. I prefer to call it "alternate constructor". Especially in the context of Python, it describes well what `@classmethod` is most useful for. And it suggests a general principle for designing your classes. Look at this complete code of the `Money` class, and I'll explain:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    @classmethod
    def from_pennies(cls, total_cents):
        dollars = total_cents // 100
        cents = total_cents % 100
        return cls(dollars, cents)
    @classmethod
    def from_string(cls, amount):
        import re
        match = re.search(r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$ ↩
            ', amount)
        if match is None:
            raise ValueError('Invalid amount: {}'.format(amount))
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
    # And then some other methods...
```

You can think of this class as having several constructors. As a general rule, you'll want to make `__init__` the most generic one, and implement the others as class methods. Sometimes, that means one of the class methods will be used more often than `__init__`. This goes against the intuitions of many Python developers; if that describes your team, you'll want to educate them in the class docs.

3.3.2 Dynamic Type: The Factory Method Pattern

This next factory pattern, called "Factory Method", is quite different. The idea is that the factory will create an object, but will choose its type from one of several possibilities, dynamically deciding at run-time based on some criteria. It's typically used when you have one base class, and are creating an object that can be one of several different derived classes.

Let's see an example. Imagine you are implementing an image processing library, creating classes to read the image from storage. So you create a base `ImageReader` class, and several derived types:

```
import abc
class ImageReader(metaclass=abc.ABCMeta):
    def __init__(self, path):
        self.path = path
    @abc.abstractmethod
    def read(self):
        pass # Subclass must implement.

    def __repr__(self):
        return '{}({})'.format(self.__class__.__name__, self.path ↩
           )

class GIFReader(ImageReader):
    def read(self):
        "Read a GIF"

class JPEGReader(ImageReader):
    def read(self):
        "Read a JPEG"

class PNGReader(ImageReader):
    def read(self):
        "Read a PNG"
```

The `ImageReader` class is marked abstract, requiring subclasses to implement the `read` method. So far, so good.

Now, when reading an image file, if its extension is ".gif", I want to use `GIFReader`. And if it is a JPEG image, I want to use `JPEGReader`. And so on. The logic is

- Analyze the file path name to get the extension,
- choose the correct reader class based on that,
- and finally create the appropriate reader object.

This is a prime candidate for automation. Let's define a little helper function:

```
def extension_of(path):
    position_of_last_dot = path.rfind('.')
    return path[position_of_last_dot+1:]
```

With these pieces, we can now define the factory:

```
# First version of get_image_reader().
def get_image_reader(path):
    image_type = extension_of(path)
    reader_class = None
    if image_type == 'gif':
        reader_class = GIFReader
    elif image_type == 'jpg':
        reader_class = JPEGReader
    elif image_type == 'png':
        reader_class = PNGReader
    assert reader_class is not None, 'Unknown extension: {}'. ↩
       format(image_type)
    return reader_class(path)
```

Classes in Python can be put in variables, just like any other object. We take full advantage here, by storing the appropriate `ImageReader` subclass in `reader_class`. Once we decide on the proper value, the last line creates and returns the reader object.

This correctly-working code is more concise, readable and maintainable than what some languages force you to go through. But in Python, we can do better. We can use the built-in dictionary type to make it even more readable and easy to maintain over time:

```
READERS = {
    'gif' : GIFReader,
    'jpg' : JPEGReader,
    'png' : PNGReader,
    }
def get_image_reader(path):
    reader_class = READERS[extension_of(path)]
    return reader_class(path)
```

Here we have a global variable mapping filename extensions to `ImageReader` subclasses. This lets us readably implement `get_image_reader` in two lines. Finding the correct class is a simple dictionary lookup, and then we instantiate and return the object. And if we support new image formats in the future, we simply add a line in the `READERS` definition. (And, of course, define its reader class.)

What if we encounter an extension not in the mapping, like `tiff`? As written above, the code will raise a KeyError. That may be what we want. Or closely related, perhaps we want to catch that, and re-raise a different exception.

Alternatively, we may want to fall back on some default. Let's create a new reader class, meant as an all-purpose fallback:

```
class RawByteReader(ImageReader):
    def read(self):
        "Read raw bytes"
```

Then you can write the factory like:

```
def get_image_reader(path):
    try:
        reader_class = READERS[extension_of(path)]
    except KeyError:
        reader_class = RawByteReader
    return reader_class(path)
```

or more briefly

```
def get_image_reader(path):
    return READERS.get(extension_of(path), RawByteReader)
```

This design pattern is commonly called the "factory method" pattern, which wins my award for Worst Design Pattern Name In History. That name (which appears to originate from a Java implementation detail) fails to tell you anything about what it's actually *for*. I myself call it the "dynamic type" pattern, which I feel is much more descriptive and useful.

3.4 The Observer Pattern

The Observer pattern defines a certain kind of "one to many" relationship. Specifically, there's one root object - let's call it the *publisher* - whose state can change in a way that's interesting to other objects. These other objects - let's call them *subscribers* - tell the publisher that they want to know when this happens.

The way they tell the publisher is to *register* (by calling a method on the publisher, which may actually be named register). Whenever this interesting state in the publisher changes, all registered subscribers are notified.

That's all pretty abstract. Let's see some concrete examples.

3.4.1 The Simple Observer

In the simplest form, each subscriber must implement a method called `update` (or something else that both sides agree on). Here's an example:

```python
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)
```

The `update` method takes a string. (It can take something else - again, so long as both publisher and subscriber agree on the interface. But we'll use a string.)

Example driver:

```
from observer1 import Publisher, Subscriber

pub = Publisher()

bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register(bob)
pub.register(alice)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

3.4.2 A Pythonic Refinement

The simple version above has a pretty standard interface. Python gives more flexibility, because you can pass functions around just like any other object. This means a subscriber can register to be notified by calling a method other than `update` - or even a completely separate function.

Regardless of whether this is a method or a function, let’s just call it a "callback". This callback should have a compatible signature, of course.

```python
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback == None:
            callback = getattr(who, 'update')
        self.subscribers[who] = callback
    def unregister(self, who):
        del self.subscribers[who]
    def dispatch(self, message):
        for subscriber, callback in self.subscribers.items():
            callback(message)
```

```python
from observer2 import *

pub = Publisher()
bob = SubscriberOne('Bob')
alice = SubscriberTwo('Alice')
john = SubscriberOne('John')

pub.register(bob, bob.update)
pub.register(alice, alice.receive)
pub.register(john)

pub.dispatch("It's lunchtime!")
pub.unregister(john)
pub.dispatch("Time for dinner")
```

3.4.3 Several Channels

So far, we've assumed the publisher only has one kind of thing to say… one kind of state that can change, which is of interest to subscribers. But what if there are several?

```
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self, channels):
        # maps channel names to subscribers
        # str -> dict
        self.channels = { channel : dict()
                          for channel in channels }
    def subscribers(self, channel):
        return self.channels[channel]
    def register(self, channel, who, callback=None):
        if callback == None:
            callback = getattr(who, 'update')
        self.subscribers(channel)[who] = callback
    def unregister(self, channel, who):
        del self.subscribers(channel)[who]
    def dispatch(self, channel, message):
        for subscriber, callback in self.subscribers(channel). ↩
           items():
            callback(message)
```

```python
from observer3 import *

pub = Publisher(['lunch', 'dinner'])
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)

pub.dispatch("lunch", "It's lunchtime!")
pub.dispatch("dinner", "Dinner is served")
```

3.5 Magic Methods

Suppose we want to create a class to work with angles, in degrees. We want this class to help us with some standard bookkeeping:

- An angle will be at least zero, but less than 360.
- If we create an angle outside this range, it automatically wraps around to an equivalent, in-range value.
- In fact, we want the conversion to happen in a range of situations:
  - If we add 270º and 270º, it evaluates to 180º instead of 540º.
  - If we subtract 180º from 90º, it evaluates to 270º instead of -90º.
  - If we multiply an angle by a real number, it wraps the final value into the correct range.
- And while we're at it, we want to enable all the other behaviors we normally want with numbers: comparisons like "less than" and "greater or equal than" or "==" (i.e., equals); division (which doesn't normally require casting into a valid range, if you think about it); and so on.

Let's see how we might approach this, by creating a basic Angle class:

```
class Angle:
    def __init__(self, value):
        self.value = value % 360
```

The modular division in the constructor is kind of neat: if you reason through it with a few positive and negative values, you'll find the math works out correctly whether the angle is overshooting or undershooting. This meets one of our key criteria already: the angle is normalized to be from 0 up to 360. But how do we handle addition? We of course get an error if we try it directly:

```
>>> Angle(30) + Angle(45)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Angle' and 'Angle'
>>>
```

We can easily define a method called `add` or something, which will let us write code like `angle3 = angle1.add(angle2)`. But that means we can't reuse the familiar syntax everyone learned in school for math. I don't know about you, but when I'm hard at work developing, I already have more than enough to learn and remember. So when possible, I prefer to rely on something so ingrained and automatic, that I'm free to focus my mental energy on more important things.

Well, great news: Python lets us do that, through a collection of object hooks called *magic methods*. It lets you define classes so that their instances can be used with all of Python's primitive operators:

- You can do all arithmetic using the normal operators: + - * / // and more
- They can be compared for equality (==) and inequality (!=)
- . . . as well as richer comparisons (< > >= <=)
- Higher-level concepts like exponentiation, absolute value, etc.
- Bit-shifting operations

Few classes will need all of these, but sometimes it's invaluable to have them available. Let's see how they can improve our `Angle` type.

3.5.1 Simple Math Magic

The pattern for each method is the same. For a given operation - say, addition - there is a special method name that starts and begins with double-underscores. For addition, it's `__add__` - the others also have sensible names. All you have to do is define that method, and instances of your class can be used with that operator.

For operations like addition that take two values and return a third, the signature looks like this:

```
    def __add__(self, other):
        return 42 # Or whatever the correct total is.
```

The first argument needs to be called "self", because this is Python. The other one does not have to be called "other", but often it is, because it has a clear meaning and is easy to remember. For our `Angle` class, we could implement it like this:

```
    def __add__(self, other):
        return Angle(self.value + other.value)
```

This lets us use the normal addition operator for arithmetic:

```
>>> total = Angle(30) + Angle(45)
>>> total.value
75
```

There are similar operators for subtraction (`__sub__`), multiplication (`__mul__`), and more.

3.5.2 Printing and Logging

There's something missing, though: what if we try to add two angles directly, without setting to a variable?

```
>>> Angle(30) + Angle(45)
<__main__.Angle object at 0x106df9198>
```

The `__add__` method is returning a correct object. But when we print it, the representation isn't so useful. It tells us the type, and the hex object ID. But what we might really want to know is the value of the angle.

There are two magic methods that can help. The first is `__str__`, which is used when printing a result:

```
    def __str__(self):
        return '{} degrees'.format(self.value)
```

The `print` function uses this, as well as `str`, and the string formatting operations:

```
>>> print(Angle(30) + Angle(45))
75 degrees
>>> print('{}'.format(Angle(30) + Angle(45)))
75 degrees
>>> str(Angle(135))
'135 degrees'
```

Sometimes, you want a string representation that is more precise, which might be at odds with the goals of a human-friendly representation. A good example is when you have several subclasses (e.g., imagine `PitchAngle` and `YawAngle` in some kind of aircraft-related library), and want to easily log the exact type and arguments needed to recreate the object. Python provides a second magic method for this purpose, called `__repr__`:

```
    # An okay implementation.
    def __repr__(self):
        return 'Angle({})'.format(self.value)
```

You access this by calling either the `repr` built-in function (think of it as working like `str`, but invokes `__repr__` instead of `__str__`), or by passing the `!r` conversion to the formatting string:

```
>>> print('{!r}'.format(Angle(30) + Angle(45)))
Angle(75)
```

The rule of thumb is that the output of `__repr__` is something that can be passed to `eval()` to recreate the object exactly. While not enforced by the language, this convention is officially recommended, and very widely followed among experienced Python programmers. It's not always practical for every class. But often it is, and it can be very useful for logging and other purposes.

3.5.3 All Things Being Equal

Another thing we want to be able to do is compare two `Angle` objects. The most basic is equality and inequality. The former is provided by `__eq__`, which should return True or False:

```
    def __eq__(self, other):
        return self.value == other.value
```

If defined, this method is used by the == operator to determine equality:

```
>>> Angle(3) == Angle(3)
True
>>> Angle(7) == Angle(1)
False
```

By default, the == operator for objects is based off the object ID, which is safe but often not very useful:

```
>>> class BadAngle:
...     def __init__(self, value):
...         self.value = value
...
>>> BadAngle(3) == BadAngle(3)
False
```

The != operator has its own magic method, `__ne__`. It works the same way:

```
    def __ne__(self, other):
        return self.value != other.value
```

What happens if you don't implement `__ne__`? In Python 3, if you define `__eq__` but not `__ne__`, then the != operator will use `__eq__`, negating the output. Especially for simple classes like `Angle`, this default behavior is logically valid. So in this case, we don't need to define a `__ne__` method at all. For more complex types, the concepts of equality and inequality may have more subtle nuances, and you will need to implement both.

3.5.3.1 Python 2 != Python 3

The story is different in Python 2. In that universe, if `__eq__` is defined but `__ne__` is not, then != does *not* use `__eq__`. Instead, it relies on the default comparison based on object ID:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...     def __eq__(self, other):
...         return self.value == other.value
...
>>> BadAngle(3) != BadAngle(3)
True
```

You will probably never actually want this behavior (which is why it was changed in Python 3). So for Python 2, if you do define `__eq__`, be sure to define `__ne__` also:

```
# A good default __ne__ for Python 2.
# This is basically what Python 3 does automatically.
    def __ne__(self, other):
        return not self.__eq__(other)
```

3.5.4 Comparisons

Now that we've covered strict equality and inequality, what's left are the fuzzier comparison operations; less than, greater than, and so on. Python's documentation calls these "rich comparison" methods, so you can feel wealthy when using them:

- `__lt__` for "less than" (<)
- `__le__` for "less than or equal" (<=)
- `__gt__` for "greater than" (>)
- `__ge__` for "greater than or equal" (>=)

For example:

```
    def __gt__(self, other):
        return self.value > other.value
```

Now the greater-than operator works correctly:

```
>>> Angle(100) > Angle(50)
True
```

Similar with `__ge__`, `__lt__`, etc. If you don't define these, you get an error, at least in Python 3:

```
>>> BadAngle(8) > BadAngle(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: BadAngle() > BadAngle()
```

`__gt__` and `__lt__` are reflections of each other. What that means is that, in many cases, you only have to define one of them. Suppose you implement `__gt__` but not `__lt__`, then do this:

```
>>> a1 = Angle(3)
>>> a2 = Angle(7)
>>> a1 < a2
True
```

This works thanks to some just-in-time introspection the Python runtime does. The `a1 < a2` is, semantically, equivalent to `a1.__lt__(a2)`. If `Angle.__lt__` is indeed defined, that semantic equivalent is executed, and the expression evaluates to its return value.

For normal scalar numbers, `n < m` is true if and only if `m > n`. For this reason, if `__lt__` does not exist, but `__gt__` does, then Python will rewrite the angle comparison: `a1.lt(a2)` becomes `a2.gt(a1)`. This is then evaluated, and the expression `a1 < a2` is set to its return value.

Note there are situations where this is actually *not* what you want. Imagine a `Point` type, for example, with two coordinates, x and y. You want `point1 < point2` to be `True` if and only if `point1.x < point2.x`, AND `point1.y < point2.y`. Similarly for `point1 > point2`. There are many points for which both `point1 < point2` and `point1 > point2` should both evaluate to `False`.

For types like this, you will want to implement both `__gt__` and `__lt__`. (Ditto for `__ge__` and `__le__`.) You might also need to raise `NotImplemented` in the method. This built-in exception signals to the Python runtime that the operation is not supported, at least for these arguments.

3.5.4.1 Shortcut: functools.total_ordering

The `functools` module in the standard library defines a class decorator named `total_ordering`. In practice, for any class which needs to implement all the rich comparison operations, using this labor-saving decorator should be your first choice.

In essence: in your class, you define both `__eq__` and **one** of the comparison magic methods: `__lt__`, `__le__`, `__gt__`, or `__ge__`. (You can define more than one, but it's not necessary.) Then you apply the decorator to the *class*:

```
import functools
@functools.total_ordering
class Angle:
    # ...
    def __eq__(self, other):
        return self.value == other.value
    def __gt__(self, other):
        return self.value > other.value
```

When you do this, all missing rich comparison operators are supplied, defined in terms of `__eq__` and the one operator you defined. This can save you a fair amount of typing, and it's worthwhile to use it if it makes sense.

There are a few situations where you won't want to use `total_ordering`. One is if the comparison logic for the type is not well-behaved enough that each operator can be inferred from the other, via straightforward boolean logic. The `Point` class is an example of this, as might some types if what you are implementing boils down to some kind of abstract algebra engine.

The other reasons not to use it are (1) performance, and (2) the more complex stack traces it generates are more trouble than they are worth. Generally, though, I recommend you assume these are *not* a problem until proven otherwise. It's entirely possible you will never encounter one of the involved stack traces. And the relatively inefficient implementations that `total_ordering` provides are unlikely to be a problem unless they are used deep inside some nested loop.

3.5.5 Python 2

As mentioned, in Python 3, if you don't define `__lt__`, and then try to compare two objects with the < operator, you get a `TypeError`. And likewise for `__gt__` and the others. That's a *very* good thing. In Python 2, you instead get a default ordering based off the object ID. This can lead to truly infuriating bugs:

```
# Python 2.
>>> class BadAngle(object):
...     def __init__(self, value):
...         self.value = value
...
>>>
>>> BadAngle(6) < BadAngle(5)
True
>>> BadAngle(6) < BadAngle(5)
False
```

What the heck just happened? When parsing and running the first `BadAngle(6) < BadAngle(5)` line, the Python runtime created two `BadAngle` instances. It turns out the left-hand object was created with an ID whose value happens to be less than that of the right-hand object. So the expression evaluates as `True`. In the second line, the opposite happened: the right-hand object won the race, so to speak, so the expression evaluates as `False`.

Horrifying race conditions like this are not your friend. Until you can upgrade to Python 3, be vigilant.

3.5.6 More Magic

The full list of numeric magic methods is listed and documented at https://docs.python.org/3/-reference/datamodel.html#emulating-numeric-types. Most of the magic methods we covered relate to numeric operations, but there are others as well. You can read more about them in the surrounding sections. They include:

- Boolean operations, like `and`, `or` and `not`
- Higher-level math operations like `abs`, exponents, and negation
- Bit-shifting operations

Index