Python: The Next Level

Aaron Maxwell

aaron@powerfulpython.com

Contents

Chapter 1

Advanced Functions

In this chapter, we go beyond the basics of using functions. I'll assume you can 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

Sometimes you want to define a function that can take any number of arguments. Python's syntax for doing that 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. It's 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.

Notice how different this is 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 lets you use 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.

My examples above above that I always called that tuple argument `*args`. That is a well-followed convention, but not required. In fact, it can be any valid identifier 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 they have a reason to name it something else. (This seems to be pretty strongly ingrained; more than once, I tried to abbreviate it `*a`, only to have my code reviewer demand I change it to `*args`.) But if you can improve readability by giving it a different name, don't hesitate to do so.

Argument Unpacking

The star modifier works in the other direction too. And the exciting thing is, 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 the complete and utter lack of any 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)
```

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)
```

These work, but are not ideal. Primarily, 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 how `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 (i.e. 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()`:[1]

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

```
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 in practice 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
```

Arguments after `**kwargs` form a special category, called *keyword-only arguments*.[2] If present, then whenever that function is called, all must specified as key-value pairs, *after* the non-keyword arguments:

```
>>> def read_data_from_files(*paths, data_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,  ↩
   data_format="json")
>>> commodities_data = read_data_from_files("commodities.xml",  ↩
   data_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:

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

```
>>> 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, or an instance of a class you create. The implications of this are profound. It lets you do certain *very* useful things with functions. It's one of those secrets that separate average Python developers from great ones, because of the *extremely* powerful abstractions that follow. **Once you get this, it can change the way you write software forever.** Best of all, these advanced patterns for using functions in Python largely transfer to other languages you will use in the future.

To explain what we mean, 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'
```

What's the nature of the error here? Since the objects in `nums` are strings, `max` compares each element lexicographically.[3] By that criteria, "7" is greater than "30". So it's not a bug, per se. The only problem is that `max` is evaluating the element by a different criteria than what we want.

The `max` algorithm is simple, so 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'
```

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

This gives us what we want: it returns the element in the original list which is maximal, as evaluated by our criteria. Now imagine we are working with different data, and 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
```

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`. Can you spot it?

It's 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, the `max` functions are all identical.

I don't know about you, but having nearly-identical functions like this drives me nuts. The way out is to realize that the comparison is based on a value that's *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`.[4] 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:

[4] 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 often actual functions, but they can be any callable.

```
>>> 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
```

Notice we are passing in the function object itself - `int` and `abs`. We are not invoking them. In other words, we don't write `int()` or `abs()` - that's a common sticking point for people new to functional programming. We are passing the function itself, which means we write `int`, not `int()`. This function object is what's used to calculate the derived value:

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

For sorting the students by GPA, we need a function that extracts 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 now be realizing just how powerful this programming pattern 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 assign functions to 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 how `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 the `max` function is customizable. You can pass in 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. To demonstrate, 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 that `itemgetter` is a function that creates and returns a function - in 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. These are especially 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]
```

Chapter 2

Decorators

Python supports a powerful tool called the *decorator*. Decorators let you add rich features to groups of functions and classes, without modifying them at all; untangle distinct, frustratingly intertwined concerns in your code, in ways not possible with vanilla functions and objects; and build powerful, extensible software frameworks. Best of all, the decorators you create are effortless for other developers to re-use... greatly amplifying the potential impact of the code you write.

You have probably used decorators before, or at least seen them. 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 the argument to print: `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 method of `app` called `route` is a decorator, applied here to the function called `hello`. This is how you specify that an HTTP GET request to the root URL ("/") is handled by the `hello` function.

Once upon a time, I implemented a Python class with a group of methods that could not be run at the same time; I had to ensure that if any method was invoked, it would not start until all others in the group had returned. And I wanted to be able to easily add new methods to that group in the future. So I created a decorator called `withlock` to enforce this - requiring no changes within the methods themselves:

```
    @withlock
    def first_method_in_group(self, arg):
        ...
    @withlock
    def another_method_in_group(self, arg):
        ...
```

Using decorators is simple and easy; even someone new to programming can learn the syntax quickly. Our objective here is different: to give you the ability to write your own decorators, in many different useful forms. This won't just greatly improve the quality of your own code. It literally has the potential to benefit your *career*, as you create decorators that others can trivially reuse to solve otherwise vexing problems.

This is worth your while. Let's get started.[1]

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

2.1 The Basic Function Decorator

The syntax for using decorators is actually a kind of shorthand. When you see this:

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

It's equivalent to this:

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

`some_decorator` is a function that (a) takes a function as an argument, and (b) returns a function. To keep us from going insane, let's define some terminology:

- The **decorator** is what comes after the `@`. It's a function.
- The **wrapped function** is the one `def`'ed on the next line. It is, obviously, also a function.
- The result of decorating a function is the **wrapper function**. It's what you actually call in your code.

Sometimes we'll also say "decorated function" to mean the wrapped function - i.e., the function that is decorated. (But be careful: some blog authors use that phrase to refer to what we call the wrapper function.)

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 that takes 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 that logs a message to stdout every time the wrapped 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. In the global space, this is then assigned the name `foo`, replacing the wrapped 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)
```

Run it and you get this:

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

At a high level, the body of `printlog` does one thing: defined a function called `wrapper`. (And then return it. Okay, two things.) Many decorators will fit this high-level pattern. 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 (wrapped) function.

Once you've applied a decorator, the original, wrapped function isn't directly accessible anymore; you can't call it in your code. The name of the function now applies to the wrapper version. But that wrapper function still retains a reference to the original, wrapped function, allowing you to call it indirectly.

There's one big shortcoming with this version of `printlog`, 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. Since `baz` has two, when the wrapper function is called, the whole thing blows up. There's no reason `printlog` must only work with unary functions; all it's doing is printing the function name. The solution is to declare `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
```

The big win here is maintainability. A decorator written this way, using variable arguments, will potentially work with functions and methods written years later - code the original developer never could have anticipated. 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 if you have a reason.

2.2 Encapsulating Data In Decorators

State can be accumulated inside the decorator function itself. Here's an example that counts how many times a function has been called:

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

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

Here we have to use the `nonlocal` keyword.[2] This is similar to `global`, except it only reaches enclosing scope. (Not the top-level, global scope). The `nonlocal` line lets us modify the value of the `count` variable defined on the first line of `countcalls`.

We generally don't have to use `nonlocal` for container types, however. Here's an example that logs a message when a function is called with a given argument for the first time:

```
def announcenew(func):
    all_args = set()
    def wrapper(n):
        if n not in all_args:
            all_args.add(n)
            print('New argument: {}'.format(n))
        return func(n)
    return wrapper
```

It works like this:

[2] `nonlocal` exists only in Python 3, and was never backported to 2. In Python 2, we would need to create a global dictionary named something like `counts` at the top level, mapping `func` objects to the integer count. Note how `nonlocal` lets us encapsulate things better. Perhaps a better alternative is to use a class-based decorator, explained later in this chapter.

```
>>> @announcenew
... def double(n):
...     return n * 2
...
>>> double(3)
New argument: 3
6
>>> double(7)
New argument: 7
14
>>> double(3)
6
>>> double(3)
6
```

The variable `all_args` itself is never assigned a new value; we are only invoking its `add` method. We never put `all_args` on the left hand side of an assignment statement, so there is no problem. In contrast above, `count` *is* assigned to a new value, because `count += 1` is just shorthand for `count = count + 1`.

2.3 Decorators For Classes

Decorators can be applied to classes, just like functions and methods. Instead of a wrapper function, the decorator should return 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 function decorators? With those, the decorator returned a new, different function object.

Another strategy for class decorators 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'>
```

Notice the object type isn't obviously related to the decorated class. That can make debugging harder, create unruly log messages, and have other unexpected effects. For this reason, I recommend you prefer the first approach.

Class decorators are often used 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
```

2.4 Decorators That Take Arguments

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

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

This is different from any decorator we've implemented so far, because it actually takes an argument. How do we write decorators that can do this? For example, maybe we have a family of decorators that add 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.

2.4.1 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.

2.5 Class-based Decorators

So far, you've seen how to implement decorators using functions. Another approach uses classes instead. The secret to making this work is the magic method `__call__`.

(Note carefully, there is a big difference between class-based decorators, and decorators that apply to classes (what we call "class decorators" above). These are completely independent concepts.)

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!'
```

This `__call__` method provides a very different way to implement decorators. Instead of passing `func` as the argument to a decorator *function*, you can pass it to the *constructor* of a decorator *class*:

```
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)
```

```
>>> @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
```

Class-based decorators give a few advantages that function-based ones lack. For one thing, the decorator is a class, which means you can leverage inheritance. This enables some interesting code-reuse patterns:

```
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 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
```

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
```

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.

2.6 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 that relies 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 truly 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.[3] Decorators created using the `wrapt` module work in situations that cause normal decorators to break, and behave correctly when used with more exotic Python language features.

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

While it's true that using `functools.wraps` on all your decorators will prevent certain problems, doing this 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 takes extra time to write the code, 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.)

For these reasons, 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.[4]

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

[4] 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`.

Chapter 3

Classes and Objects: Beyond The Basics

This chapter assumes you are familiar with the basic syntax of defining classes, methods, and inheritance. We will look beyond the basics, to some practical useful tools and patterns.

Like with any object-oriented language, it's useful to learn about the **design patterns** that apply - reusable solutions to common problems involving classes and objects. A LOT of material has been written on this topic since the turn of the century, and even before. Curiously, though, much of that doesn't exactly apply to Python - or, at least, applies in a different way.

The reason has to do with language features Python has, which languages like Java, C++ and C# lack. (Or have lacked, until relatively recently.) Many of the well-known design patterns become far simpler in Python: dynamic typing, functions as first-class objects, and some other additions to the object model all mean design patterns just work differently in this language.

3.1 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 using a decorator named `property`.

One common use is to create a special "read-only" attribute, which doesn't correspond to an actual data member. 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. This is one use of properties: if you deliberately want to have an attribute that others can read, but not write to.

However, the property protocol lets us also define a way to support writes. Here's how we do that:

```
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)
```

When set to, it invokes the setter method:

```
>>> bob = Person("Bob", "Johnson")
>>> bob.first_name
'Bob'
>>> bob.last_name
'Johnson'
>>> bob.full_name = "Bobby Jacobson"
>>> bob.first_name
'Bobby'
>>> bob.last_name
'Jacobson'
```

The first time I saw this, I had all sorts of questions. "Wait, why is the full_name method defined twice? And why is the second decorator named after its target? How on earth does this even byte compile?"

It's actually correct, and designed to work this way. The `@property def full_name` must come first. After that, in the namespace of the class, a method named `full_name.setter` exists, which is used as a decorator for the next `def full_name`. 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.)

As mentioned above, another use of properties is to create read-only attributes. Often there is a hidden member attribute behind it:

```
class Ticket:
    def __init__(self, price, serial_number):
        self.price = price
        self._serial_number = serial_number
    @property
    def serial_number(self):
        return self._serial_number
```

This allows the class itself to modify the value internally, but prevent outside code from doing so. (Assuming it follows the Python convention of not accessing underscore attributes.)

The idea of properties exists in several modern languages. One idiom that's important in languages like Java, but turns out to be unnecessary in Python, looks like this:

```
class SomeClass:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
```

A very similar example is even in Python's documentation of the `property` function. In a language like Java, this is often a good idea: it gives you the freedom to change the underlying

private attribute in the future, without disrupting any user of the class. However, in Python, it's really unnecessary. A better way is to not use a property at all, at least at first:

```
class SomeClass:
    x = 0
```

Now suppose, at some point in the future, it becomes necessary to put some kind of bounds-checking on valid set values. At that point you can refactor to store the value in a hidden member, and do the check in a setter:

```
class SomeClass:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        # x must be nonnegative.
        assert value >= 0, value
        self._x = value
```

In Java, doing this could cause a ton of trouble, because other code - perhaps many different external projects - might have relied on reading and writing the attribute directly. This is why in Java, often you will want to proactively define `getX()` and `setX()` methods, even if at first they are not doing anything interesting. In Python, however, there is no possible shortcoming, because the *interface* is the same. Even though setting to `x` actually executes a setter function, the part of the program doing that doesn't know or care.

3.2 The Factory Pattern

The idea of the factory pattern is to provide a relatively simple way to create a complex object. Here's a very trivial example:

```
class A:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

def create_a(n):
    return A(n, n+1, n+2)
```

The whole point of a factory is to simplify how you create objects. There are two very different forms of factories:

- Where the object's type is fixed, but we want to have several different ways to create it. This is called a *Simple Factory*.
- 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 two in Python.

3.2.1 Alternative Constructors: The Simple Factory

Imagine a simple class representing US dollars:

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

The 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 we will be modeling a giant jar of pennies. This is where a factory function can be useful:

```
# Factory function that takes a single argument, returning
# an appropriate Money instance.
def money_from_pennies(num_pennies):
    dollars, cents = divmod(num_pennies, 100)
    return Money(dollars, cents)
```

Or maybe we have the amount as a string, such as `"$12.34"`:

```
# 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 can be convenient, and worth defining. They are effectively alternate constructors: callables we can use with different arguments, which are parsed and used to create the final object.

This works, and mirrors how factories are used in other languages. (They may be static methods on the class, but Python's module system allows freestanding functions to work just as well, with less typing.)

There's a problem here, however: the return type, i.e. the `Money` class, is hard-coded. If we change the name, e.g. to `Dollars`, we can easily miss it in the refactoring. More importantly, if we subclass `Money`, there is no way to make these factory functions work with the subclass - we'd have to essentially copy and modify them.

As it turns out, Python provides a solution to these problems that is virtually absent among other languages: the `classmethod` decorator. You invoke it like this:

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

The function called `money_from_pennies` is now a method of the `Money` class, called `from_pennies`. But the big difference is the new first argument: `cls`. Here's what's going on: When applied to a method of a class, the `classmethod` decorator modifies how the method on the next line is invoked and interpreted. The first argument is no longer `self`, i.e. an instance of the class. The first argument is now *the class itself*. 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 factory methods 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.

3.2.2 Dynamic Type: The Factory Method Pattern

This form of factory works quite differently. The idea is that the factory can create an object of several different types, dynamically deciding the correct one 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 some classes that will 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):
        # Implementing these read methods is an exercise for the  ↩
            reader.
        pass

class JPEGReader(ImageReader):
    def read(self):
        pass

class PNGReader(ImageReader):
    def read(self):
        pass
```

The `ImageReader` class is marked as 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 goes like this:

- 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 sounds like 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:

```
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 assigned to variables, and used just like any other object. We are taking full advantage of that 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 works great, and is much more concise, readable and maintainable than what some languages force you to go through. But in Python, we can do even 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 defined 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 files in the future, why, that's just one more line in the `READERS` dictionary!

What happens 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 perhaps raising a different exception). Alternatively, we may want to fall back on some default. Suppose we add this reader class:

```
class RawByteReader(ImageReader):
    def read(self):
        pass
```

Then our factory can be written like this:

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

3.3 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.

3.3.1 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:

```
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")
```

3.3.2 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.

```
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)
```

```
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")
```

3.3.3 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)
```

```
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")
```

3.4 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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.3.1 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)
```

3.4.4 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 `__gte__`, `__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 to: `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.

3.4.4.1 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.

3.4.5 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.

3.4.6 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