"FEELS LIKE NEO LEARNING JIU JITSU IN THE MATRIX."

AARON MAXWELL

2ND EDITION

POWERFUL PYTHON

THE MOST IMPACTFUL PATTERNS, FEATURES AND DEVELOPMENT STRATEGIES MODERN PYTHON PROVIDES

Powerful Python

Table of Contents

- Doing More with Python
  - Python Versions
  - Python Application Environments
  - Python Package Management
- Scaling With Generators
  - Iteration in Python
  - Generator Functions
  - Generator Patterns and Scalable Composability
  - Python is Filled With Iterators
  - The Iterator Protocol
- Creating Collections with Comprehensions
  - List Comprehensions
  - Formatting For Readability (And More)
  - Multiple Sources and Filters
  - Comprehensions and Generators
  - Dictionaries, Sets, and Tuples
  - Limits of Comprehensions
- Advanced Functions
  - Accepting & Passing Variable Arguments
  - Functions As Objects

Key Functions in Python
Decorators
The Basic Decorator
Data In Decorators
Decorators That Take Arguments
Class-based Decorators
Decorators For Classes
Preserving the Wrapped Function
Exceptions and Errors
The Basic Idea
Exceptions Are Objects
Raising Exceptions
Catching And Re-raising
The Most Diabolical Python Anti-Pattern
Classes and Objects: Beyond The Basics
Quick Note on Python 2
Properties
The Factory Patterns
The Observer Pattern
Magic Methods
Rebelliously Misusing Magic Methods

Automated Testing and TDD
- What is Test-Driven Development?
- Unit Tests And Simple Assertions
- Fixtures And Common Test Setup
- Asserting Exceptions
- Using Subtests
- Final Thoughts

String Formatting
- Replacing Fields
- Number Formats (and "Format Specs")
- Width, Alignment, and Fill
- F-Strings
- Percent Formatting

Logging in Python
- The Basic Interface
- Configuring The Basic Interface
- Passing Arguments
- Beyond Basic: Loggers
- Log Destinations: Handlers and Streams
- Logging to Multiple Destinations
- Record Layout with Formatters

What’s Next?

DOING MORE WITH PYTHON

I don’t need to tell you how amazing Python is. You *know*. Or you wouldn’t be reading this book.

It’s still fun to recognize what an exciting time it is for Python. Amazon’s best-selling programming book list is *filled* with Python. Attendance at PyCon keeps climbing, on top of the regional conferences and spin-off meetups. More and more organizations are using Python… from tiny startups, to multinational corporations, to NASA and the JPL, and everything in between. Even MIT reworked their famous, venerable "Introduction to Computer Science and Programming" class, replacing Scheme with… you guessed it, Python.

But enough cheerleading.

There are *massive heaping piles* of books for people new to Python, new to programming, or both. But you’re past the point where those do you much good. If you’ve been coding in Python for a while already, or Python is your second or third or seventh programming language… you need more.

Reading blog posts can help, as can studying open-source code, and (if you can swing it) working alongside a seasoned Pythonista. But these aren’t the

most convenient ways to learn.

And that's why I wrote this book.

Python is richer than you imagine - as a language, as an ecosystem. For many engineering domains, it has grown into a truly outstanding choice for implementing high-quality, robust, maintainable software - everything from one-off scripts to sprawling, mission-critical applications. This book is designed to help you master all of that: to teach you techniques, patterns, and tools to permanently catapult your skill with everything Python has to offer.

To accomplish this, I did not hesitate to make hard choices. Understand this book is highly **opinionated**. I focus on certain topics, and certain ways of structuring code, because I believe it gives you the best payoff and value for your reading time. Many blog posts have been written about different aspects of Python development; frankly, some of them are not good advice. My goal in this book is to give you excellent advice.

To that end, this book is **practical**. Everything herein is based on the lessons learned writing real-world software, usually as part of a team of engineers. That means factors like maintainability, robustness, and readability are considered more important than anything else. There is a balance between leveraging powerful abstractions, and writing code that is easy to reason about correctly by everyone on your team. Every page of this book walks that line.

Throughout, I give much attention to cognitive aspects of development. How do you write code which you and others can reason about easily, quickly, *and*

accurately? This is one reason variable and function naming is so important. But it goes **far beyond** that, to intelligently choosing which language features and library resources to use, and which to avoid.

Thus, this book is **selective** in its topics. It's not too large, as measured by number of pages. **That's a feature, not a bug: you already have too much to read.** The focus is on what's most impactfully valuable to *you*, so that - as much as possible - everything you learn will serve you for *years*.

That's one reason this book is focused on Python 3. I'm fortunately also able to help those of you using Python 2.7, because the most valuable patterns, practices and strategies in Python are surprisingly independent of Python version. So I'll demonstrate each idea using Python 3 code, and where the syntax of Python 2.7 differs, I'll point out those differences as we go along.

And it's important that you do learn Python 3. The main Linux distributions are mostly switching to 3; third-party library support is *extremely* solid; and the list of improvements and features Python 3 has over 2.7 is huge, and getting longer.[1] This book is written to prepare you for Python's foreseeable future.

When I teach live workshops in Python for working developers, my intention is to instill transformatively powerful skills and abilities, which will serve attendees for the rest of their careers. I bring this same intention for *you*, reading this book. If you have any comments or questions, I'd love to hear them - reach me by email at aaron@powerfulpython.com.

And as a reader of this book, you will want to subscribe to the Powerful Python Newsletter,[2] because it gives you important new articles on intermediate and advanced Python. For those of you with a digital edition of this book, it's also how I let you know there's a new edition to download.

Speaking of the digital version: if you've bought this as a physical book, you can add a DRM-free digital copy for $4.99 USD. This includes digital updates to future editions; you'll get the 3rd edition when it comes out, as well as intermediate point releases. (Between the 1st and 2nd editions, digital readers got three substantial updates, with new chapters and extensive revisions.) Go to powerfulpython.com/book-upgrade to take advantage.

Python Versions

We're at an interesting transition point. I know most of you are using Python 3 at least part of the time. Many of you use only Python 2.7, and many only Python 3. In this book, I take care of everyone.

Most code examples in this book are written for Python 3.5 and later. Sometimes I'll talk about features specific to a later version - when we talk about Python 3.6's f-strings, for example - and I'll make it clear when that happens.

I've also written this book's code in a way that can be easily adapted to Python 2.7. Surprisingly, in many cases the exact same code will work in both Python 2.7 and Python 3! **The structure of quality Python software, and the patterns and strategies that work exceptionally well in the real world, are *surprisingly* independent of version.**

And of course, there are sometimes differences in syntax. So throughout the book, I include footnotes and special sub-sections that explain how things are different in Python 2.7, as they come up.

(And if you *are* working in that version, pay close attention when I show you Python 2 code. If there are two ways to do something in Python 2.7 - one of which is forward-compatible with Python 3, and another which is not - I'll *always* show you the former. It tells you how to write for Python 2 today, in a way that gives you less to re-learn when you eventually upgrade.)

People often ask if it's worth their effort to upgrade from Python 2 to 3. You'll find passionate opinions on every side of this fence. I can offer this advice: **If I had to sum up what makes Python 3 different, it's that creating high quality software is easier in Python 3.** You can, obviously, create high quality software in Python 2; you just have to work a bit harder, sometimes.

Of course, you also lose the opportunity to use some exciting and fun Python-3-only features. This arguably doesn't matter from an engineering standpoint. But it might matter to you personally. And there will come a day when having only Python 2.x on your resume will make it look dated.

In any event, this book is about *current* versions of Python. That includes Python versions 3.5 and later, as well as 2.7. I do consider Python 2.6 obsolete at this point, though you may still need to use it for legacy reasons. If so, some topics will be a bit more difficult to translate into your code base.

This book uses `str.format()` to format strings:

```
>>> "Hello, {}!".format("John")
'Hello, John!'
```

For a quick refresher, skim the first few sections of the "String Formatting" chapter. To round out this intro, we'll look at some important practical aspects of developing modern Python applications.

Python Application Environments

Python sports a concept called the *virtual environment*. It is sometimes called a "virtualenv" or a "venv". You can think of a virtual environment as a kind of lightweight container for a Python application, where the application can see its own particular set of Python libraries, at specific versions. This provides some significant benefits for deployment:

- Dependencies can be precisely tracked and specified, and even kept in version control.
- Two different Python applications with conflicting dependencies can peacefully coexist on the same machine.
- Python packages can be installed without requiring elevated system privileges.

How you create a virtual environment depends on whether you're using Python 3 or 2. For Python 3, you invoke `python3 -m venv`, adding one more argument - the name of a folder to create:

```
# The recommended method in Python 3.
$ python3 -m venv webappenv
```

(The `$` is the shell prompt.) For the near future, you can also use the `pyvenv` command. This works fine, but is deprecated, and scheduled to disappear in Python 3.8:

```
# Does the same thing, but is deprecated.
$ pyvenv webappenv
```

What these both do is create a folder named "webappenv" in the current directory. The Python 2 tool, `virtualenv`, has the same interface; wherever you see `python -m venv` in this book, you can substitute `virtualenv` instead:

```
# Same as the above, in Python 2.
$ virtualenv webappenv
```

Regardless of which you use, you'll end up with a new folder named `webappenv`, containing all sorts of goodies. To access them, you must run a script called "activate". In macOS and Linux, type the following in your shell:

```
$ source webappenv/bin/activate
(webappenv)$
```

For Windows, run the script `webappenv\Scripts\activate.bat` instead:

```
C:\labs> webappenv\Scripts\activate.bat
(webappenv) C:\labs>
```

Notice your prompt has changed: now it contains `(webappenv)`. The script called `activate` did not start a new shell or session; all it did was alter your PATH and PYTHONPATH environment variables, and a few others (like the PS1 variable, which specifies how your shell renders the prompt.) You just *activated* your virtual environment, as we say. When it's active for that particular shell session, we say you are working *in the virtual environment*.

Suppose your system Python executable is at `/usr/bin/python3` (or `/usr/bin/python` for version 2). And suppose you've created the

webappenv folder in `/Users/sam/mywebapp/webappenv`. With the virtual environment activated, you have your own local copy of the Python interpreter. You can check this with the `which` command on macOS and Linux:

```
(webappenv)$ which python
/Users/sam/mywebapp/webappenv/bin/python
(webappenv)$ python -V
Python 3.6.0
```

Or the `where` command on Windows:

```
(webappenv) C:\labs> where python
C:\labs\webappenv\Scripts\python.exe
C:\Python36\python.exe
```

This is for the virtual environment we created with `pyvenv`, which means it's Python 3. Now, what if you want to restore your old PATH (and PYTHONPATH, etc.)? Within the virtualenv, you now have a function defined called `deactivate`:

```
(webappenv)$ deactivate
$ which python
/usr/bin/python
```

Now imagine you are writing a Python application: let's call it `mediatag`, one designed for tagging files in your media collection. It's implemented in Python 2.7. We can do the following:

```
$ cd /Users/sam/mediatag
$ virtualenv mediatagenv
$ source mediatagenv/bin/activate
(mediatagenv)$ which python
```

```
/Users/sam/mywebapp/mediatagenv/bin/python
(mediatagenv)$ python -V
Python 2.7.13
```

This shows one minor benefit of virtual environments: it provides a new way to control the precise version of Python used by an application. But the main benefit for using virtual environments has to do with resolving requirements, upgrades, and dependencies with packages and libraries.

Python Package Management

The sordid history of Python library and package management is full of hidden twists, perilous turns, and dark corners hiding sinister beasts. Good news: you don't have to worry about any of that.

Modern Python provides an application called `pip`, which allows you to easily install third-party Python libraries and applications. It incorporates many of the lessons learned from its predecessors, sidestepping problems that previously had to be dealt with manually. And it works very well with Python virtual environments.

The first step is to install `pip`. With Python 3, this is included for you automatically, and is installed in your virtual environment:

```
$ source venv/bin/activate
(venv)$ python -V
Python 3.6.0
(venv)$ which pip
/Users/sam/myapp/venv/bin/pip
```

For Python 2, if `pip` is already installed on the system, your virtual environment will be created to include it. For macOS and Windows, recent versions of Python 2 automatically include `pip`; if not, you can quickly find out how to install it by searching online.

Once you have the `pip` executable, you can use `pip install` to install libraries just within the virtual environment. For example, `requests` is a high-quality HTTP library. Install it like so:

```
pip install requests
```

This is the `install` command. You will see some output, narrating the process of installing `requests` at a specific version. Once complete, you will be able to open a Python prompt and `import requests`.

The `pip install` command is also used to upgrade packages. For example, sometimes a fresh virtual environment may install a slightly stale version of `pip`. `pip` is just another package, so you can upgrade it with the `-U` or `--upgrade` option:

```
pip install --upgrade pip
```

Installed packages are, by default, fetched from Pypi - the official online Python package repository. Any package or library listed at https://pypi.python.org/pypi can be installed with `pip`. You can uninstall them with `pip uninstall`.

Now, some of these packages' files are substantial, or even compiled into object files. You definitely don't want to put them in version control. How do you register the exact version dependencies your app has for third-party libraries? And how do you manage upgrades (and even downgrades) over time?

`pip` provides a good solution for this. The first part of it relies on the `pip freeze` command:

```
(venv)$ pip freeze
requests==2.7.0
```

This prints the packages installed from Pypi, one per line, with the exact `version`. What you can do is place this in a file named `requirements.txt`:

```
(venv)$ pip freeze > requirements.txt
(venv)$ cat requirements.txt
requests==2.7.0
```

This file is what you will check into version control. You can recreate the application environment, right down to the minor versions of each dependent library, simply by passing `requirements.txt` to `pip`. Whether your coworker is fetching the raw source to start development, or if the devops team sets up a CI environment that runs the automated tests, the environment is consistent and well-defined, from development to staging to production.

You can pass any file path to `python3 -m venv` (and `pyvenv`, and `virtualenv`). For organizational convenience, many choose to put it in the top-level folder of the repository holding the Python application. There are two schools of thought on what to name it.

One school picks a consistent name, which is used for every project. "venv" is very popular:

```
python3 -m venv venv
```

The idea is that every Python project will have a folder in its top level called `venv` to contain the virtual environment. This has several advantages. For one, you can easily activate the virtual environment for any application, just

by typing `source venv/bin/activate`. In fact, you can define a shell alias to help:

```
# Type "venv" <enter> to get in the virtual environment.
alias venv='source venv/bin/activate'
```

You can also configure your version control system to ignore any folder named "venv", and thereby avoid ever accidentally committing your virtual environment. (You don't want to do that. It's a lot of files, and will annoy your fellow developers.)

The other naming scheme is to give it a name that has something to do with the application. For example, for an application called "mywebapp", you might create it like this:

```
python3 -m venv mywebappenv
```

The advantage of this is that, when activated, the prompt is modified to tell you which particular virtual environment your shell prompt is using. This can be helpful if you work with many different Python applications, as it's much more informative than a prompt that just says "(venv)".

The downside comes from the inconsistency of the folder name: keeping the folder out of version control is more error-prone, and activating the virtual environment requires the distraction of conscious thought each time. Both approaches are valid; it really comes down to which you and your teammates like better.

1 See https://powerfulpython.com/blog/whats-really-new-in-python-3/ for a summary.

2 https://powerfulpython.com/python-newsletter/

SCALING WITH GENERATORS

This `for` loop seems simple:

```
for item in items:
    do_something_with(item)
```

And yet, miracles hide here. As you probably know, the act of efficiently going through a collection, one element at a time, is called *iteration*. But few understand how Python's iteration system really works… how deep and well-thought-out it is. This chapter makes you one of those people, giving you the ability to naturally write **highly scalable** Python applications… able to handle ever-larger data sets in performant, memory-efficient ways.

Iteration is also core to one of Python's most powerful tools: the *generator function*. Generator functions are not just a convenient way to create useful iterators. They enable some exquisite patterns of code organization, in a way that - by their very nature - intrinsically encourage excellent coding habits.

This chapter is special, because understanding it threatens to make you a permanently better programmer *in every language*. Mastering Python generators tends to do that, because of the distinctions and insights you gain along the way. Let's dive in.

Iteration in Python

Python has a built-in function called `iter()`. When you pass it a collection, you get back an *iterator object*:

```
>>> numbers = [7, 4, 11, 3]
>>> iter(numbers)
<list_iterator object at 0x10219dc50>
```

Just as in other languages, a Python iterator produces the values in a sequence, one at a time. You probably know an iterator is like a moving pointer over the collection:

```
>>> numbers_iter = iter(numbers)
>>> for num in numbers_iter: print(num)
7
4
11
3
```

You don't normally need to do this. If you instead write `for num in numbers`, what Python effectively does under the hood is call `iter()` on that collection. This happens automatically. Whatever object it gets back is used as the iterator for that `for` loop:

```
# This...
for num in numbers:
    print(num)

# ... is effectively just like this:
numbers_iter = iter(numbers)
for num in numbers_iter:
    print(num)
```

An iterator over a collection is a separate object, with its own identity - which you can verify with `id()`:

```
>>> # id() returns a unique number for each object.
... # Different objects will always have different IDs.
>>> id(numbers)
4330133896
>>> id(numbers_iter)
4330216640
```

How does `iter()` actually get the iterator? It can do this in several ways, but one relies on a magic method called `__iter__`. This is a method any class (including yours) may define; when called with no arguments, it must return a fresh iterator object. Lists have it, for example:

```
>>> numbers.__iter__
<method-wrapper '__iter__' of list object at 0x10130e4c8>
>>> numbers.__iter__()
<list_iterator object at 0x1013180f0>
```

Python makes a distinction between objects which are *iterators*, and objects which are *iterable*. We say an object is *iterable* if and only if you can pass it to `iter()`, and get a ready-to-use iterator. If that object has an `__iter__` method, `iter()` will call it to get the iterator. Python lists and tuples are iterable. So are strings, which is why you can write `for char in my_str:` to iterate over `my_str`'s characters. Any container you might use in a `for` loop is iterable.

A `for` loop is the most common way to step through a sequence. But sometimes your code needs to step through in a more fine-grained way. For this, use the built-in function `next()`. You normally call it with a single

argument, which is an iterator. Each time you call it, `next(my_iterator)` fetches and returns the next element:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> # Create a fresh iterator...
... names_it = iter(names)
>>> next(names_it)
'Tom'
>>> next(names_it)
'Shelly'
>>> next(names_it)
'Garth'
```

What happens if you call `next(names_it)` again? `next()` will raise a special built-in exception, called `StopIteration`:

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

This is part of Python's *iterator protocol*. Raising this specific exception is, by design, how an iterator signals the sequence is done. You rarely have to raise or catch this exception yourself, though we'll see some patterns later where it's useful to do so. A good mental model for how a `for` loop works is to imagine it calling `next()` each time through the loop, exiting when `StopIteration` gets raised.

When using `next()` yourself, you can provide a second argument, for the default value. If you do, `next()` will return that instead of raising `StopIteration` at the end:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> new_names_it = iter(names)
>>> next(new_names_it, "Rick")
'Tom'
>>> next(new_names_it, "Rick")
'Shelly'
>>> next(new_names_it, "Rick")
'Garth'
>>> next(new_names_it, "Rick")
'Rick'
>>> next(new_names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(new_names_it, "Jane")
'Jane'
```

Now, let's consider a different situation. What if you aren't working with a simple sequence of numbers or strings, but something more complex? What if you are calculating or reading or otherwise obtaining the sequence elements as you go along? Let's start with a simple example (so it's easy to reason about). Suppose you need to write a function creating a list of square numbers, which will be processed by other code:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But there is potential problem lurking here. Can you spot it?

Here's one: what if `MAX` is not 5, but 10,000,000? Or 10,000,000,000? Or more? Your memory footprint is pointlessly dreadful: the code here creates a *massive* list, uses it *once*, then throws it away. On top of that, the second `for` loop cannot event *start* until the entire list of squares has been fully calculated. If some poor human is using this program, they'll wonder if the program is stuck.

Even worse: What if you aren't doing arithmetic to get each element - which is fast and cheap - but making a truly expensive calculation? Or making an API call over the network? Or reading from a database? Your program is sluggish, even unresponsive, and might even crash with an out-of-memory error. Its users will think you're a terrible programmer.

The solution is to create an iterator to start with, lazily computing each value only when needed. Then each cycle through the loop happens just in time.

For the record, here is how you create an equivalent iterator class, which fully complies with Python's iterator protocol:

```
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value
```

```
# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Holy crap, that's horrible. There's got to be a better way.

Good news: there's a better way. It's called a **generator function**, and you're going to love it!

Generator Functions

Python provides a tool called the **generator function**, which… well, it's hard to describe everything it gives you in one sentence. Of its many talents, I'll first focus on how it's a *very* useful shortcut for creating iterators.

A generator function looks a lot like a regular function. But instead of saying `return`, it uses a new and different keyword: `yield`. Here's a simple example:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

Use it in a for loop like this:

```
>>> for num in gen_nums():
...     print(num)
0
1
2
3
```

Let's go through and understand this. When you call `gen_nums()` like a function, it immediately returns a **generator object**:

```
>>> sequence = gen_nums()
>>> type(sequence)
<class 'generator'>
```

The *generator function* is `gen_nums` - what we define and then call. A function is a generator function if and only if it uses "yield" instead of "return". The *generator object* is what that generator function returns when called - `sequence`, in this case. A generator function will *always* return a generator object; it can't return anything else. And this generator object is an iterator, which means you can iterate through it using `next()` or a `for` loop:

```
>>> sequence = gen_nums()
>>> next(sequence)
0
>>> next(sequence)
1
>>> next(sequence)
2
>>> next(sequence)
3
>>> next(sequence)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> # Or in a for loop:
... for num in gen_nums(): print(num)
...
0
1
2
3
```

The flow of code works like this: when `next()` is called the first time, or the `for` loop first starts, the body of `gen_nums` starts executing at the beginning, returning the value to the right of the `yield`.

So far, this is much like a regular function. But the next time `next()` is called - or, equivalently, the next time through the `for` loop - the function doesn't start at the beginning again. It starts on the line *after the yield statement*. Look at the source of `gen_nums()` again:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

`gen_nums` is more general than a function or subroutine. It's actually a *coroutine*. You see, a regular function can have several exit points (otherwise known as `return` statements). But it has only one entry point: each time you call a function, it always starts at the first line of the function body.

A coroutine is like a function, except it has several possible *entry* points. It starts with the first line, like a normal function. But when it "returns", the coroutine isn't exiting, so much as *pausing*. Subsequent calls with `next()` - or equivalently, the next time through the `for` loop - start at that `yield` statement again, right where it left off; the re-entry point is the line after the `yield` statement.

And that's the key: **Each `yield` statement simultaneously defines an exit point, *and* a re-entry point.**

For generator objects, each time a new value is requested, the flow of control picks up on the line after the `yield` statement. In this case, the next line increments the variable `n`, then continues with the while loop.

Notice we do not raise `StopIteration` anywhere in the body of `gen_nums()`. When the function body finally exits - after it exits the while loop, in this case - the generator object automatically raises `StopIteration`.

Again: each `yield` statement simultaneously defines an exit point, *and* a re-entry point. In fact, you can have multiple `yield` statements in a generator:

```
def gen_extra_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
    yield 42 # Second yield
```

Here's the output when you use it:

```
>>> for num in gen_extra_nums():
...     print(num)
0
1
2
3
42
```

The second `yield` is reached after the `while` loop exits. When the function reaches the implicit return at the end, the iteration stops. Reason through the code above, and convince yourself it makes sense.

Let's revisit the earlier example, of cycling through a sequence of squares. This is how we first did it:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

As an exercise, pause here, open up a new Python file, and see if you can write a `gen_squares` generator function that accomplishes the same thing.

Done? Great. Here's what it looks like:

```
>>> def gen_squares(max_num):
...     for num in range(max_num):
...         yield num ** 2
...
>>> MAX = 5
>>> for square in gen_squares(MAX):
...     print(square)
0
1
4
9
16
```

Now, this `gen_squares` has a problem in Python 2, but not Python 3. Can you spot it?

Here it is: `range` returns an iterator in Python 3, but in Python 2 it returns a list. If MAX is huge, that creates a huge list inside, killing scalability. So if you are using Python 2, your `gen_squares` needs to use `xrange` instead, which acts just like Python 3's `range`.

The larger point here affects all versions of Python. Generator functions *potentially* have a small memory footprint, but only if you code intelligently. When writing generator functions, be watchful for hidden bottlenecks.

Now, strictly speaking, we don't *need* generator functions for iteration. We just *want* them, because they make certain patterns of scalability far easier. Now that we're in a position to understand it, let's look at the `SquaresIterator` class again:

```
# Same code we saw earlier.
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Each value is obtained by invoking its `__next__` method, until it raises `StopIteration`. This produces the same output; but look at the source for the `SquaresIterator` class, and compare it to the source for the generator above. Which is easier to read? Which is easier to maintain? And when requirements change, which is easier to modify without introducing errors? Most people find the generator solution easier and more natural.

Authors often use the word "generator" by itself, to mean either the generator function, *or* the generator object returned when you call it. Typically the writer thinks it's obvious by the context which they are referring to; sometimes it is, sometimes not. Sometimes the writer is not even clear on the distinction to begin with. But it's important: just as there is a big difference between a function, and the value it returns when you call it, so is there a big difference between the generator function, and the generator object it returns.

In your own thought and speech, I encourage you to only use the phrases "generator function" and "generator object", so you are always clear inside yourself, and in your communication. (Which also helps your teammates be more clear.) The only exception: when you truly mean "generator functions and objects", lumping them together, then it's okay to just say "generators". I'll lead by example in this book.

Generator Patterns and Scalable Composability

Here's a little generator function:

```
def matching_lines_from_file(path, pattern):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

`matching_lines_from_file()` demonstrates several important practices for modern Python, and is worth studying. It does simple substring matching on lines of a text file, yielding lines containing that substring.

The first line opens a read-only file object, called `handle`. If you haven't been opening your file objects using `with` statements, start today. The main benefit is that once the `with` block is exited, the file object is automatically closed - even if an exception causes a premature exit. It's similar to:

```
try:
    handle = open(path)
    # read from handle
finally:
    handle.close()
```

(The `try`/`finally` is explained in the exceptions chapter.) Next we have `for line in handle`. This useful idiom, which not many people seem to know about, is a special case for text files. Each iteration through the `for` loop, a new line of text will be read from the underlying text file, and placed in the `line` variable.

Sometimes people foolishly take another approach, which I have to warn you about:

```
# Don't do this!!
for line in handle.readlines():
```

`.readlines()` (plural) reads in the *entire file*, parses it into lines, and returns a list of strings - one string per line. By now, you realize how this destroys the generator function's scalability.

Another approach you will sometimes see, which *is* scalable, is to use the file object's `.readline()` method (singular), which manually returns lines one at a time:

```
# .readline() reads and returns a single line of text,
# or returns the empty string at end-of-file.
line = handle.readline()
while line != '':
    # do something with line
    # ...
    # At the end of the loop, read the next line.
    line = handle.readline()
```

But simply writing `for line in handle` is clearer and easier.

After that, it's straightforward: matching lines have any trailing `\n`-character stripped, and are yielded to the consumer. When writing generator functions, you want to ask yourself "what is the maximum memory footprint of this function, and how can I minimize it?" You can think of scalability as inversely proportional to this footprint. For `matching_lines_from_file()`, it will be about equal to the size of

the longest line in the text file. So it's appropriate for the typical human-readable text file, whose lines are small.

(It's also possible to point it to, say, a ten-terabyte text file consisting of exactly one line. If you expect something like *that*, you'll need a different approach.)

Now, suppose a log file contains lines like this:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
INFO: Sent email campaign, completed normally
WARNING: Almost out of beer
```

... and you exercise `matching_lines_from_file()` like so:

```
for line in matching_lines_from_file("log.txt","WARNING:"):
    print(line)
```

That yields these records:

```
WARNING: Disk usage exceeding 85%
WARNING: Almost out of beer
```

Suppose your application needs that data in dict form:

```
{"level": "WARNING", "message": "Disk usage exceeding 85%"}
{"level": "DEBUG", "message":
    "User 'tinytim' upgraded to Pro version"}
```

We want to scalably transform the records from one form to another - from strings (lines of the log file), to Python dicts. So let's make a new generator function to connect them:

```
def parse_log_records(lines):
    for line in lines:
        level, message = line.split(": ", 1)
        yield {"level": level, "message": message}
```

Now we can connect the two:

```
# log_lines is a generator object
log_lines = matching_lines_from_file("log.txt", "WARNING:")
for record in parse_log_records(log_lines):
    # record is a dict
    print(record)
```

Of course, `parse_log_records()` can be used on its own:

```
with open("log.txt") as handle:
    for record in parse_log_records(handle):
        print(record)
```

`matching_lines_from_file()` and `parse_log_records()` are like building blocks. Properly designed, they can be used to build different data processing streams. I call this *scalable composability*. It goes beyond designing composable functions and types. Ask yourself how you can make the components scalable, **and** whatever is assembled out of them scalable too.

Let's discuss a particular design point. Both `matching_lines_from_file()` and `parse_log_records()` produce an iterator. (Or, more specifically, a generator object). But they have a discrepancy on the input side: `parse_log_records()` accepts an iterator, but `matching_lines_from_file()` requires a path to a file to read from. This means `matching_lines_from_file()` combines two

functions: read lines of text from a file, then filter those lines based on some criteria.

Combining functions like this is often what you want in realistic code. But when designing components to flexibly compose together, inconsistent interfaces like this can be limiting. Let's break up the services in `matching_lines_from_file()` into two generator functions:

```
def lines_from_file(path):
    with open(path) as handle:
        for line in handle:
            yield line.rstrip('\n')

def matching_lines(lines, pattern):
    for line in lines:
        if pattern in line:
            yield line
```

You can compose these like so:

```
lines = lines_from_file("log.txt")
matching = matching_lines(lines, 'WARNING:')
for line in matching:
    print(line)
```

Or even redefine `matching_lines_from_file()` in terms of them:

```
def matching_lines_from_file(pattern, path):
    lines = lines_from_file(path)
    matching = matching_lines(lines, pattern)
    for line in matching:
        yield line
```

Conceptually, this factored-out `matching_lines` does a *filtering* operation; all lines are read in, and a subset are yielded.

`parse_log_records()` is different. One input record (a `str` line) maps to exactly one output record (a `dict`). Mathematically, it's a *mapping* operation. Think of it as a transformer or adapter. `lines_from_file()` is in a third category; instead of taking a stream as input, it takes a completely different parameter. Since it still returns an iterator of records, think of it as a *source*. And any real program will eventually want to do something with that stream, consuming it without producing another iterator; call that a *sink*.

You need all these pieces to make a working program. When designing a chainable set of generator functions like this - or even better, a toolkit for constructing internal data pipelines - ask yourself whether each component is a sink, a source, or whether it does filtering, or mapping; or whether it's some combination of these. Just asking yourself this question leads to a more usable, readable, and maintainable codebase. And if you're making a library which others will use, you're more likely to end up with a toolkit so powerfully flexible, people use it to build programs you never imagined.

I want you to notice something about `parse_log_records()`. As I said, it fits in the "mapping" category. And notice its mapping is one-to-one: one line of text becomes one dictionary. In other words, each record in the input - a `str` - becomes *exactly one* record in the output - a `dict`.

That isn't always the case. Sometimes, your generator function needs to consume several input records to create one output record. Or the opposite: one input record yields several output records.

Here's an example of the latter. Imagine a text file containing lines in a poem:[1]

```
all night our room was outer-walled with rain
drops fell and flattened on the tin roof
and rang like little disks of metal
```

Let's create a generator function, `words_in_text()`, producing the words one at a time. First approach:

```
# lines is an iterator of text file lines,
# e.g. returned by lines_from_file()
def words_in_text(lines):
    for line in lines:
        for word in line.split():
            yield word
```

This generator function takes a *fan out* approach. No input records are dropped, which means it doesn't do any filtering; it's still purely in the "mapping" category of generator functions. But the mapping isn't one to one. Rather, one input record produces one or more output records. So, when you run the following code:

```
poem_lines = lines_from_file("poem.txt")
poem_words = words_in_text(poem_lines)
for word in poem_words:
    print(word)
```

... it produces this output:

```
all
night
our
room
was
outer-walled
...
```

That first input record - "all night our room was outer-walled with rain" - yields eight words (output records). Ignoring any blank lines in the poem, every line of prose will produce at least one - probably several - words.

The idea of fanning out is interesting, but simple enough. It's more complex when we go the opposite direction: fanning *in*. That means the generator function consumes more than one input record to produce each output record. Doing this successfully requires an awareness of the input's structure, and you'll typically need to encode some simple parsing logic.

Imagine a text file containing residential house sale data. Each record is a set of key-value pairs, one pair per line, with records separated by blank lines:

```
address: 1423 99th Ave
square_feet: 1705
price_usd: 340210

address: 24257 Pueblo Dr
square_feet: 2305
price_usd: 170210

address: 127 Cochran
square_feet: 2068
price_usd: 320500
```

To read this data into a form usable in our code, what we want is a generator function - let's name it `house_records()` - which accepts a sequence of strings (lines) and parses them into convenient dictionaries:

```
>>> lines_of_house_data = lines_from_file("housedata.txt")
>>> houses = house_records(lines_of_house_data)
>>> # Fetch the first record.
... house = next(houses)
>>> house['address']
```

```
'1423 99th Ave'
>>> house = next(houses)
>>> house['address']
'24257 Pueblo Dr'
```

How would you create this? If practical, pause here, open up a code editor, and see if you can implement it.

Okay, time's up. Here is one approach:

```
def house_records(lines):
    record = {}
    for line in lines:
        if line == '':
            yield record
            record = {}
            continue
        key, value = line.split(': ', 1)
        record[key] = value
    yield record
```

Notice where the `yield` keywords appear. The last line of the `for` loop reads individual key-value pairs. Starting with an empty record dictionary, it's populated with data until `lines` produces an empty line. That signals the current record is complete, so it's `yield`-ed, and a new record dictionary created. The end of the very last record in `housedata.txt` is signaled not by an empty line, but by the end of the file; that's why we need the final `yield` statement.

As defined, `house_records()` is a bit clunky if we're normally reading from a text file. It makes sense to define a new generator function accepting just the path to the file:

```
def house_records_from_file(path):
    lines = lines_from_file(path)
    for house in house_records(lines):
        yield house

# Then in your program:
for house in house_records_from_file("housedata.txt"):
    print(house["address"])
```

You may have noticed many of these examples have a bit of boilerplate, when one generator function internally calls another. The last two lines of `house_records_from_file` say:

```
    for house in house_records(lines):
        yield house
```

Python 3 provides a shortcut, which lets you accomplish this in one line, with the `yield from` statement:

```
def house_records_from_file(path):
    lines = lines_from_file(path)
    yield from house_records(lines)
```

Even though "yield from" is two words, semantically it's like a single keyword, and distinct from `yield`. The `yield from` statement is used specifically in generator functions, when they yield values directly from another generator object (or, equivalently, by calling another generator function). Using it often simplifies your code, as you see in `house_records_from_file()`. Going back a bit, here's how it works with `matching_lines_from_file()`:

```
def matching_lines_from_file(pattern, path):
    lines = lines_from_file("log.txt")
    yield from matching_lines(lines, 'WARNING:')
```

The formal name for what `yield from` does is "delegating to a sub-generator", and instills a deeper connection between the containing and inner generator objects. In particular, generator objects have certain methods - `send`, `throw` and `close` - for passing information back *into* the context of the running generator function. I won't cover them in this edition of the book, as they are not currently widely used; you can learn more by reading PEPs 342 and 380.[2] If you do use them, `yield from` becomes necessary to enable the flow of information back into the scope of the running coroutine.

Python is Filled With Iterators

Let's look at Python 3 dictionaries:[3]

```
>>> calories = {
...     "apple": 95,
...     "slice of bacon": 43,
...     "cheddar cheese": 113,
...     "ice cream": 15, # You wish!
... }
>>> items = calories.items()
>>> type(items)
<class 'dict_items'>
```

So what is this `dict_items` object returned by `calories.items()`? It turns out to be what Python calls a *view*. There is not any kind of base view type; rather, an object quacks like a dictionary view if it supports three things:

- `len(view)` returns the number of items,
- `iter(view)` returns an iterator over the key-value pairs, and
- `(key, value) in view` returns True if that `key-value` pair is in the dictionary, else False.

In other words, a dictionary view is iterable, with a couple of bonus features. It also dynamically updates if its source dictionary changes:

```
>>> items = calories.items()
>>> len(items)
4
>>> calories['orange'] = 50
>>> len(items)
5
```

```
>>> ('orange', 50) in items
True
>>> ('orange', 20) in items
False
```

Dictionaries also have `.keys()` and `.values()`. Like `.items()`, they each return a view. But instead of key-value pairs, they only contain keys or values, respectively:

```
>>> foods = calories.keys()
>>> counts = calories.values()
>>> 'yogurt' in foods
False
>>> 100 in counts
False
>>> calories['yogurt'] = 100
>>> 'yogurt' in foods
True
>>> 100 in counts
True
```

In Python 2 (explained more below), `items()` returns a list of key-value tuples, rather than a view; Python 2 also has an `iteritems()` method that returns an iterator (rather than an iterable view object). Python 3's version of the `items()` method essentially obsoletes both of these. When you do need a list of key-value pairs in Python 3, just write `list(calories.items())`.

Iteration has snuck into many places in Python. The built-in `range` function returns an iterable:

```
>>> seq = range(3)
>>> type(seq)
<class 'range'>
```

```
>>> for n in seq: print(n)
0
1
2
```

The built-in `map`, `filter`, and `zip` functions all return iterators:

```
>>> numbers = [1, 2, 3]
>>> big_numbers = [100, 200, 300]
>>> def double(n):
...     return 2 * n
>>> def is_even(n):
...     return n % 2 == 0
>>> mapped = map(double, numbers)
>>> mapped
<map object at 0x1013ac518>
>>> for num in mapped: print(num)
2
4
6
```

```
>>> filtered = filter(is_even, numbers)
>>> filtered
<filter object at 0x1013ac668>
>>> for num in filtered: print(num)
2
```

```
>>> zipped = zip(numbers, big_numbers)
>>> zipped
<zip object at 0x1013a9608>
>>> for pair in zipped: print(pair)
(1, 100)
(2, 200)
(3, 300)
```

Notice that `mapped` is something called a "map object", rather than a list of the results of the calculation; and similar for `filtered` and `zipped`. These

are all iterators - giving you all the benefits of iteration, built into the language.

Python 2's Differences

For Python 2, I'll start with some recommendations, before explaining them:

- With dictionaries, always use `viewitems()` rather than `items()` or `iteritems()`. The only exception: if you truly need a list of tuples, use `items()`.
- Likewise for `viewkeys()` and `viewvalues()`, rather than `keys()`, `iterkeys()`, `values()`, and `itervalues()`.
- Use `xrange()` instead of `range()`, unless you have a special need for an actual list.
- Be aware that `map`, `filter`, and `zip` create lists; if your data may grow large, use `imap`, `ifilter` or `izip` from the `itertools` module instead.

These differently named methods and functions essentially have the same behavior as Python 3's more scalable versions. Python 2's `xrange` is just like Python 3's `range`; Python 2's `itertools.imap` is just like Python 3's `map`; and so on.

Let's examine Python 2's dictionary methods. In Python 2, `calories.items()` returns a list of `(key, value)` tuples. So if the dictionary has 10,000 keys, you'd get a list of 10,000 tuples. Similarly, `calories.keys()` returns a list of keys; `calories.values()` returns

a list of values. The problems with this will be obvious to you by now: the loop blocks until you create and populate a list, which is immediately thrown away once the loop exits.

Python 2 addressed this by introducing two other methods: `iteritems()`, returning an iterator over the key-value tuples; and (later) `viewitems()`, which returned a view - an iterable type. Similarly, `keys()` gave a list of keys, and they added `iterkeys()` and then `viewkeys()`; and again for `values()`, `itervalues()`, and `viewvalues()`.

In Python 3, what used to be called `viewitems()` was renamed `items()`, and the old `items()` and `iteritems()` went away. Similarly, `keys()` and `values()` were changed to return views instead of lists.

For your own Python 2 code, I recommend you start using `viewitems()`, except when you have an explicit reason to do otherwise. Using `iteritems()` is certainly better than using `items()`, and for Python 2 code, generally works just as well as `viewitems()`. However, if you ever decide to upgrade that codebase with `2to3`, the resulting code will be closer to your original program.[4] Python 2's `viewitems` basically obsoleted `iteritems`, which is why the latter has no equivalent in Python 3.

The situation with `range` is simpler. Python 2's original `range()` function returned a list; later, `xrange()` was added, returning an iterable, and practically speaking obsoleting Python 2's `range()`. But many people continue to use `range()`, for a range (ha!) of reasons. Python 3's version of `range()` is essentially the same as Python 2's `xrange()`, and Python 3

has no function named `xrange`. (Of course, `list(range(…))` will give you an actual list, if you need it.)

`map`, `filter`, and `zip` are well used in certain circles. If you want your Python 2 code using these functions to be fully forward-compatible, you have to go to a little more trouble: their iterator-equivalents are all in the `itertools` module. So, instead of this:

```
mapped = map(double, numbers)
```

you will need to write this:

```
from itertools import imap
mapped = imap(double, numbers)
```

The `2to3` program will convert Python 2's `imap(f, items)` to `map(f, items)`, but will convert Python 2's `map(f, items)` to `list(map(f, items))`. The `itertools` module similarly has `ifilter` and `izip`, for which the same patterns apply.

It's important to realize that everything described for Python 3 also applies to Python 2.7, *if* you use the different names of the relevant methods and functions. And that is what I recommend you do, so you get the scalability benefits of iterators, and have an easier transition to Python 3.

The Iterator Protocol

This optional section explains Python's **iterator protocol** in formal detail, giving you a precise and low-level understanding of how generators, iterators, and iterables all work. For the day-to-day coding of most programmers, it's not nearly as important as everything else in this chapter. That said, you need this information to implement your own, custom iterable collection types. Personally, I also find knowing the protocol helps me reason through iteration-related issues and edge cases; by knowing these details, I'm able to quickly troubleshoot and fix certain bugs that might otherwise eat up my afternoon. If this all sounds valuable to you, keep reading; otherwise, feel free to skip to the next chapter.

As mentioned, Python makes a distinction between *iterators*, versus objects that are *iterable*. The difference is subtle to begin with, and frankly it doesn't help that the two words sound nearly identical. Keep clear in your mind that "iterator" and "iterable" are distinct but related concepts, and the following will be easier to understand.

Informally, an iterator is something you can pass to `next()`, or use exactly once in a `for` loop. More formally, an object in Python 3 is an iterator if follows the **iterator protocol**. And an object follows the iterator protocol if it meets the following criteria:

- It defines a method named `__next__`, called with no arguments.
- Each time `__next__()` is called, it produces the next item in the sequence.

- Until all items have been produced. Then, subsequent calls to `__next__()` raise `StopIteration`.
- It also defines a boilerplate method named `__iter__`, called with no arguments, and returning the same iterator. Its body is literally `return self`.

Any object with these methods can call itself a Python iterator. You are not intended to call the `__next__()` method directly. Instead, you will use the built-in `next()` function. To understand better, here is a simplified way you might write your own `next()` function:

```
def next(it, default=None):
    try:
        return it.__next__()
    except StopIteration:
        if default is None:
            raise
        return default
```

(This version will not let you specify `None` as a default value; the real built-in `next()` will. Otherwise, it's essentially accurate.)

All the above has one difference in Python 2: the iterator's method is named `.next()` rather than `.__next__()`. Abstracting over this difference is one reason to use the built-in, top-level `next()` function. Of course, using `next()` lets you specify a default value, whereas `.__next__()` and `.next()` do not.

Now, all the above is for the "iterator". Let's explain the other word, "iterable". Informally, an object is *iterable* if it knows how to create an

iterator over its contents, which you can access with the built-in `iter()` function. More formally, a Python container object is *iterable* if it meets one of these two criteria:

- It defines a method called `__iter__()`, which creates and returns an iterator over the elements in the container; or
- it follows the *sequence protocol*. This means it defines `__getitem__` - the magic method for square brackets - and lets you reference `foo[0]`, `foo[1]`, etc., raising an `IndexError` once you go past the last element.

(Notice "iterator" is a noun, while "iterable" is usually an adjective. This can help you remember which is which.)

When implementing your own container type, you probably want to make it iterable, so you and others can use it in a `for` loop. Depending on the nature of the container, it's often easiest to implement the sequence protocol. As an example, consider this simple `UniqueList` type, which is a kind of hybrid between a list and a set:

```
class UniqueList:
    def __init__(self, items):
        self.items = []
        for item in items:
            self.append(item)
    def append(self, item):
        if item not in self.items:
            self.items.append(item)
    def __getitem__(self, index):
        return self.items[index]
```

Use it like this:

```
>>> u = UniqueList([3,7,2,9,3,4,2])
>>> u.items
[3, 7, 2, 9, 4]
>>> u[3]
9
>>> u[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in __getitem__
IndexError: list index out of range
```

The `__getitem__` method implements square-bracket access; basically, Python translates `u[3]` into `u.__getitem__(3)`. We've programmed this object's square brackets to operate much like a normal list, in that the initial element is at index 0, and you get subsequent elements with subsequent integers, not skipping any. And when you go past the end, it raises `IndexError`. If an object has a `__getitem__` method behaving in just this way, `iter()` knows how to create an iterator over it:

```
>>> u_iter = iter(u)
>>> type(u_iter)
<class 'iterator'>
>>> for num in u_iter: print(num)
3
7
2
9
4
```

Notice we get a lot of this behavior for free, simply because we're using an actual list internally (and thus delegating much of the `__getitem__` logic to it). That's a clue for you, whenever you make a custom collection that acts like a list - or one of the other standard collection types. If your object

internally stores its data in one of the standard data types, you'll often have an easier time mimicking its behavior.

Implementing the sequence protocol - i.e., writing a `__getitem__` method which acts like a list's - is one way to make your class iterable. The other involves writing an `__iter__` method. When called with no arguments, it must return some object which follows the iterator protocol, described above. In the worst case, you'll need to implement something like the `SquaresIterator` class from earlier in this chapter, with its own `__next__` and `__iter__` methods. But usually you don't have to work that hard, because you can simply return a generator object instead. That means `__iter__` is a generator function itself, or it internally calls some other generator function, returning its value.

Both iterables and iterators must have an `__iter__` method. Both are called with no argument, and both return an iterator object. The only difference: the one for the iterator returns its `self`, while an iterable's will create and return a *new* iterator. And if you call it twice, you get two different iterators.

This similarity is intentional, to simplify control code that can accept either iterators or iterables. Here's the mental model you can safely follow: when Python's runtime encounters a `for` loop, it will start by invoking `iter(sequence)`. This *always* returns an iterator: either `sequence` itself, or (if `sequence` is only iterable) the iterator created by `sequence.__iter__()`.

Iterables are everywhere in Python. Almost all built-in collection types are iterable: `list`, `tuple`, and `set`, and even `dict` (though more often you'll want to use `dict.items()` or `dict.viewitems()`). In your own custom collection classes, sometimes the easiest way to implement `__iter__()` actually involves using `iter()`. For instance, this will not work:

```
class BrokenInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        return self.items
```

If you try it, you get a `TypeError`:

```
>>> items = BrokenInLoops()
>>> for item in items:
...     print(item)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iter() returned non-iterator of type 'list'
```

It doesn't work because `__iter__()` is supposed to return an iterator, but a list object is *not* an iterator; it is simply *iterable*. You can fix this with a one-line change: use `iter()` to create an iterator object inside of `__iter__()`, and return that object:

```
class WorksInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        # This class is identical to BrokenInLoops,
        # except for this next line.
        return iter(self.items)
```

This makes `WorksInLoops` itself iterable, because `__iter__` returns an actual iterator object - making `WorksInLoops` follow the iterator protocol correctly. That `__iter__` method generates a fresh iterator each time:

```
>>> items = WorksInLoops()
>>> for item in items:
...     print(item)
7
3
9
```

1 From "Summer Rain", by Amy Lowell. https://www.poets.org/poetsorg/poem/summer-rain

2 https://www.python.org/dev/peps/pep-0342/ and https://www.python.org/dev/peps/pep-0380/

3 If you're more interested in Python 2, follow along. Every concept in this section fully applies to Python 2.7, with syntax differences we'll discuss at the end.

4 `2to3` will replace both `iteritems()` and `viewitems()` with `items()`; but the precise semantics of the converted program will more closely match your Python 2 code if you use `viewitems()` to begin with.

CREATING COLLECTIONS WITH COMPREHENSIONS

A *list comprehension* is a high level, declarative way to create a list in Python. They look like this:

```
>>> squares = [ n*n for n in range(6) ]
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

This is exactly equivalent to the following:

```
>>> squares = []
>>> for n in range(6):
...     squares.append(n*n)
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice that in the first example, what you type is declaring *what* kind of list you want, while the second is specifying *how* to create it. That's why we say it is high-level and declarative: it's as if you are stating what kind of list you want created, and then let Python figure out how to build it.

Python lets you write other kinds of comprehensions other than lists. Here's a simple dictionary comprehension, for example:

```
>>> blocks = { n: "x" * n for n in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

This is exactly equivalent to the following:

```
>>> blocks = dict()
>>> for n in range(5):
...     blocks[n] = "x" * n
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

The main benefits of comprehensions are readability and maintainability. Most people find them *very* readable; even developers who have never encountered them before can usually correctly guess what it means. And there is a deeper, cognitive benefit: once you've practiced with them a bit, you will find you can write them with very little mental effort - keeping more of your attention free for other tasks.

Beyond lists and dictionaries, there are several other forms of comprehension you will learn about it in this chapter. As you become comfortable with them, you will find them to be versatile and very Pythonic - meaning, you'll find they fit well into many other Python idioms and constructs, lending new expressiveness and elegance to your code.

List Comprehensions

A list comprehension is the most widely used and useful kind of comprehension, and is essentially a way to create and populate a list. Its structure looks like:

```
[ EXPRESSION for VARIABLE in SEQUENCE ]
```

EXPRESSION is any Python expression, though in useful comprehensions, the expression typically has some variable in it. That variable is stated in the *VARIABLE* field. *SEQUENCE* defines the source values the variable enumerates through, creating the final sequence of calculated values.

Here's the simple example we glimpsed earlier:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice the result is just a regular list. In `squares`, the expression is `n*n`; the variable is `n`; and the source sequence is `range(6)`. The sequence is a `range` object; in fact, it can be any iterable… another list or tuple, a generator object, or something else.

The expression part can be anything that reduces to a value:

- Arithmetic expressions like `n+3`
- A function call like `f(m)`, using `m` as the variable

- A slice operation (like `s[::-1]`, to reverse a string)
- Method calls (`foo.bar()`, iterating over a sequence of objects)
- And more.

Some complete examples:

```
>>> # First define some source sequences...
... pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> # And a helper function...
... def repeat(s):
...     return s + s
...
>>> # Now, some list comprehensions:
... [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]
>>> [ 10 - x for x in numbers ]
[1, 11, 14, -10, -1, 13]
>>> [ pet.lower() for pet in pets ]
['dog', 'parakeet', 'cat', 'llama']
>>> [ "The " + pet for pet in sorted(pets) ]
['The cat', 'The dog', 'The llama', 'The parakeet']
>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Notice how all these fit the same structure. They all have the keywords "for" and "in"; those are required in Python, for any kind of comprehension you may write. These are interleaved among three fields: the expression; the variable (i.e., the identifier from which the expression is composed); and the source sequence.

The order of elements in the final list is determined by the order of the source sequence. But you can filter out elements by adding an "if" clause:

```
>>> def is_palindrome(s):
...     return s == s[::-1]
...
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> words = ["bib", "bias", "dad", "eye", "deed", "tooth"]
>>>
>>> [ n*2 for n in numbers if n % 2 == 0 ]
[-8, 40]
>>>
>>> [pet.upper() for pet in pets if len(pet) == 3]
['DOG', 'CAT']
>>>
>>> [n for n in numbers if n > 0]
[9, 20, 11]
>>>
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

The structure is

```
[ EXPR for VAR in SEQUENCE if CONDITION ]
```

where *CONDITION* is an expression that evaluates to `True` or `False`, depending on the variable.[1] Note that it can be either a function applied to the variable (`is_palindrome(word)`), or something more complex (`len(pet) == 3`). Choosing to use a function can improve readability, and also let you apply filter logic whose code won't fit in one line.

A list comprehension must always have the "for" word, even if the beginning expression is just the variable itself. For example, when we say:

```
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

Sometimes people think `word for word in words` seems redundant (it does), and try to shorten it… but that doesn't work:

```
>>> [word in words if is_palindrome(word)]
  File "<stdin>", line 1
    [word in words if is_palindrome(word)]
                                          ^
SyntaxError: invalid syntax
>>>
```

Formatting For Readability (And More)

Realistic list comprehensions tend to be too long to fit nicely on a single line. And they are composed of distinct logical parts, which can vary independently as the code evolves. This creates a couple of inconveniences, which are solved by a very convenient fact: *Python's normal rules of whitespace are suspended inside the square brackets*. You can exploit this to make them more readable and maintainable, splitting them across multiple lines:

```
def double_short_words(words):
    return [ word + word
             for word in words
             if len(word) < 5 ]
```

Another variation, which some people prefer:

```
def double_short_words(words):
    return [
        word + word
        for word in words
        if len(word) < 5
        ]
```

What I've done here is split the comprehension across separate lines. You can, and should, do this with any substantial comprehension. It's great for several reasons, the most important being the instant gain in readability. This comprehension has three separate ideas expressed inside the square brackets: the expression (`word + word`); the sequence (`for word in words`); and the filtering clause (`if len(word) < 5`). These are logically separate aspects, and by splitting them across different lines, it takes less

cognitive effort for a human to read and understand than the one-line version. It's effectively pre-parsed for you, as you read the code.

There's another benefit: version control and code review diffs are more pin-pointed. Imagine you and I are on the same development team, working on this code base in different feature branches. In my branch, I change the expression to `"word + word + word"`; in yours, you change the threshold to `"len(word) < 7"`. If the comprehension is on one line, version control tools will perceive this as a merge conflict, and whoever merges last will have to manually fix it.[2] But since this list comprehension is split across three lines, our source control tool can automatically merge both our branches. And if we're doing code reviews like we should be, the reviewer can identify the precise change immediately, without having to scan the line and think.

Multiple Sources and Filters

You can have several `for VAR in SEQUENCE` clauses. This lets you construct lists based on pairs, triplets, etc., from two or more source sequences:

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
 'orange doll', 'purple bike', 'purple basketball',
 'purple skateboard', 'purple doll', 'pink bike',
 'pink basketball', 'pink skateboard', 'pink doll']
```

Every pair from the two sources, `colors` and `toys`, is used to calculate a value in the final list. That final list has 12 elements, the product of the lengths of the two source lists.

I want you to notice that the two `for` clauses are independent of each other; `colors` and `toys` are two unrelated lists. Using multiple `for` clauses can sometimes take a different form, where they are more interdependent. Consider this example:

```
>>> ranges = [range(1,7), range(4,12,3), range(-5,9,4)]
>>> [ float(num)
...   for subrange in ranges
...   for num in subrange ]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 4.0, 7.0, 10.0, -5.0,
-1.0, 3.0, 7.0]
```

The source sequence - "`ranges`" - is a list of range objects.[3] Now, this list comprehension has two `for` clauses again. But notice one depends on the other. The source of the second is the variable for the first!

It's not like the colorful-toys example, whose `for` clauses are independent of each other. When chained together this way, order matters:

```
>>> [ float(num)
...   for num in subrange
...   for subrange in ranges ]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'subrange' is not defined
```

Python parses the list comprehension from left to right. If the first clause is `for num in subrange`, at that point `subrange` is not defined. So you have to put `for subrange in ranges` *first*. You can chain more than two `for` clauses together like this; the first one will just need to reference a previously-defined source, and the others can use sources defined in the previous `for` clause, like `subrange` is defined.

Now, that's for chained `for` clauses. If the clauses are independent, does the order matter at all? It does, just in a different way. What's the difference between these two list comprehensions:

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
'orange doll', 'purple bike', 'purple basketball',
```

```
'purple skateboard', 'purple doll', 'pink bike',
'pink basketball', 'pink skateboard', 'pink doll']
>>>
>>> [ color + " " + toy
...   for toy in toys
...   for color in colors ]
['orange bike', 'purple bike', 'pink bike', 'orange
basketball', 'purple basketball', 'pink basketball',
'orange skateboard', 'purple skateboard', 'pink
skateboard', 'orange doll', 'purple doll', 'pink doll']
```

The order here doesn't matter in the sense it does for chained `for` clauses, where you *must* put things in a certain order, or your program won't run. Here, you have a choice. And that choice *does* effect the order of elements in the final comprehension. The first element in each is "orange bike". And notice the second element is different. Think a moment, and ask yourself: why? Why is the first element the same in both comprehensions? And why is it only the *second* element that's different?

It has to do with which sequence is held constant while the other varies. It's the same logic that applies when nesting regular `for` loops:

```
>>> # Nested one way...
... build_colors_toys = []
>>> for color in colors:
...     for toy in toys:
...         build_colors_toys.append(color + " " + toy)
>>> build_colors_toys[0]
'orange bike'
>>> build_colors_toys[1]
'orange basketball'
>>>
>>> # And nested the other way.
... build_toys_colors = []
>>> for toy in toys:
...     for color in colors:
```

```
...         build_toys_colors.append(color + " " + toy)
>>> build_toys_colors[0]
'orange bike'
>>> build_toys_colors[1]
'purple bike'
```

The second `for` clause in the list comprehension corresponds to the innermost `for` loop. Its values vary through their range more rapidly than the outer one.

In addition to using many `for` clauses, you can have more than one `if` clause, for multiple levels of filtering. Just write several of them in sequence:

```
>>> numbers = [ 9, -1, -4, 20, 17, -3 ]
>>> odd_positives = [
...     num for num in numbers
...     if num > 0
...     if num % 2 == 1
... ]
>>> print(odd_positives)
[9, 17]
```

Here, I've placed each `if` clause on its own line, for readability - but I could have put both on one line. When you have more than one `if` clause, each element must meet the criteria of *all* of them to make it into the final list. In other words, `if` clauses are "and-ed" together, not "or-ed" together.

What if you want to do "or" - to include elements matching at least one of the `if` clause criteria, omitting only those not matching either? List comprehensions don't allow you do to that directly. The comprehension mini-language is not as expressive as Python itself, and there are lists you might need to construct which cannot be expressed as a comprehension.

But sometimes you can cheat a bit by defining helper functions. For example, here's how you can filter based on whether the number is a multiple of 2 **or** 3:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> def is_mult_of_2_or_3(num):
...     return (num % 2 == 0) or (num % 3 == 0)
...
>>> [
...     num for num in numbers
...     if is_mult_of_2_or_3(num)
... ]
[9, -4, 20, -3]
```

We discuss this more in the "Limitations" section, later in the chapter.

You can use multiple `for` and `if` clauses together:

```
>>> weights = [0.2, 0.5, 0.9]
>>> values = [27.5, 13.4]
>>> offsets = [4.3, 7.1, 9.5]
>>>
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets
...   if offset > 5.0
...   if weight * value < offset ]
[(0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 7.1),
(0.2, 13.4, 9.5), (0.5, 13.4, 7.1), (0.5, 13.4, 9.5)]
```

The only rule is that the first `for` clause must come before the first `if` clause. Other than that, you can interleave `for` and `if` clauses in any order, though most people seem to find it more readable to group all the `for` clauses together at first, then the `if` clauses together at the end.

Comprehensions and Generators

List comprehensions create lists:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
```

When you need a list, that's great, but sometimes you don't *need* a list, and you'd prefer something more scalable. It's like the situation near the start of the generators chapter:

```
# This again.
NUM_SQUARES = 10*1000*1000
many_squares = [ n*n for n in range(NUM_SQUARES) ]
for number in many_squares:
    do_something_with(number)
```

The entire `many_squares` list must be fully created - all memory for it must be allocated, and every element calculated - before `do_something_with` is called even *once*. And memory usage goes through the roof.

You know one solution: write a generator function, and call it. But there's an easier option: write a *generator expression*. This is the official name for it, but it really should be called a "generator comprehension". Syntactically, it looks just like a list comprehension - except you use parentheses instead of square brackets:

```
>>> generated_squares = ( n*n for n in range(NUM_SQUARES) )
>>> type(generated_squares)
<class 'generator'>
```

This "generator expression" creates a *generator object*, in the exact same way a list comprehension creates a list. Any list comprehension you can write, you can use to create an equivalent generator object, just by swapping "`(`" and "`)`" for "`[`" and "`]`".

And you're creating the object directly, without having to define a generator function to call. In other words, a generator expression is a convenient shortcut when you need a quick generator object:

```
# This...
many_squares = ( n*n for n in range(NUM_SQUARES) )

# ... is EXACTLY EQUIVALENT to this:
def gen_many_squares(limit):
    for n in range(limit):
        yield n * n
many_squares = gen_many_squares(NUM_SQUARES)
```

As far as Python is concerned, there is no difference.

Everything you know about list comprehensions applies to generator expressions: multiple `for` clauses, `if` clauses, etc. You only need to type the parentheses.

In fact, sometimes you can even omit them. When passing a generator expression as an argument to a function, you will sometimes find yourself typing "`((`" followed by "`))`". In that situation, Python lets you omit the inner pair. Imagine, for example, you are sorting a list of customer email addresses, looking at only those customers whose status is "active":

```
>>> # User is a class with "email" and "is_active" fields.
... # all_users is a list of User objects.
```

```
>>> # Sorted list of active user's email addresses.
... # Passing in a generator expression.
>>> sorted((user.email for user in all_users
...          if user.is_active))
['fred@a.com', 'sandy@f.net', 'tim@d.com']
>>>
>>> # Omitting the inner parentheses.
... # Still passing in a generator expression!
>>> sorted(user.email for user in all_users
...        if user.is_active)
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

Notice how readable and natural this is (or will be, once you've practiced a bit). One thing to watch out for: you can only inline a generator expression this way when passed to a function or method of one argument. Otherwise, you get a syntax error:

```
>>>
>>> # Reverse that list. Whoops...
... sorted(user.email for user in all_users
...         if user.is_active, reverse=True)
  File "<stdin>", line 2
SyntaxError: Generator expression must be parenthesized if not
sole argument
```

Python can't unambiguously interpret what you mean here, so you must use the inner parentheses:

```
>>> # Okay, THIS will get the reversed list.
... sorted((user.email for user in all_users
...         if user.is_active), reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

And of course, sometimes it's more readable to assign the generator expression to a variable:

```
>>> active_emails = (
...         user.email for user in all_users
...         if user.is_active
... )

>>> sorted(active_emails, reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

Generator expressions without parentheses suggest a unified way of thinking about comprehensions, which link generator expressions and list comprehensions together. Here's a generator expression for a sequence of squares:

```
( n**2 for n in range(10) )
```

And here it is again, passed to the built-in `list()` function:

```
list( n**2 for n in range(10) )
```

And here it is as a list comprehension:

```
[ n**2 for n in range(10) ]
```

When you understand generator expressions, it's easy to see list comprehensions as a derivative data structure. And the same applies for dictionary and set comprehensions (covered next). With this insight, you start seeing new opportunities to use all of them in your own code, improving its readability, maintainability, and performance in the process.

Generator Expression or List Comprehension?

If generator expressions are so great, why would you use list comprehensions? Generally speaking, when deciding which to use, your code

will be more scalable and responsive if you use a generator expression. Except, of course, when you actually need a list. There are several considerations.

First, if the sequence is unlikely to be very big - and by big, I mean a minimum of thousands of elements long - you probably won't benefit from using a generator expression. That's just not big enough for scalability to matter. They're also immutable. If you need random access, or to go through the sequence twice, or you might need to append or remove elements, generator expressions won't work.

This is especially important when writing methods or functions whose return value is a sequence. Do you return a generator expression, or a list comprehension? In theory, there's no reason to ever return a list instead of a generator object; a list can be trivially created by passing it to `list()`. In practice, the interface may be such that the caller will really want an actual list. Also, if you are constructing the return value as a list within the function, it's silly to return a generator expression over it - just return the actual list.

And if your intention is to create a library usable by people who may not be advanced Pythonistas, that can be an argument for returning lists. Almost all programmers are familiar with list-like data structures. But fewer are familiar with how generators work in Python, and may - quite reasonably - get confused when confronted with a generator object.

Dictionaries, Sets, and Tuples

Just like a list comprehension creates a list, a dictionary comprehension creates a dictionary. You saw an example at the beginning of this chapter; here's another. Suppose you have this `Student` class:

```
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major
```

Given a list `students` of student objects, we can write a dictionary comprehension mapping student names to their GPAs:

```
>>> { student.name: student.gpa for student in students }
{'Jim Smith': 3.6, 'Ryan Spencer': 3.1,
 'Penny Gilmore': 3.9, 'Alisha Jones': 2.5,
 'Todd Reynolds': 3.4}
```

The syntax differs from that of list comprehensions in two ways. Instead of square brackets, you're using curly brackets - which makes sense, since this creates a dictionary. The other difference is the expression field, whose format is `"key: value"`, since a `dict` has key-value pairs. So the structure is

```
{ KEY : VALUE for VARIABLE in SEQUENCE }
```

These are the only differences. *Everything* else you learned about list comprehensions applies, including filtering with `if` clauses:

```
>>> def invert_name(name):
...     first, last = name.split(" ", 1)
...     return last + ", " + first
...
>>> # Get "lastname, firstname" of high-GPA students.
... { invert_name(student.name): student.gpa
...   for student in students
...   if student.gpa > 3.5 }
{'Smith, Jim': 3.6, 'Gilmore, Penny': 3.9}
```

You can create sets too. Set comprehensions look exactly like a list comprehension, but with curly braces instead of square brackets:

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science',
 'Economics', 'Basket Weaving']
>>> # And the same as a set:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

(How does Python distinguish between a set and dict comprehension? Because the `dict`'s expression is a key-value pair, while a `set`'s has a single value.)

What about tuple comprehensions? This is fun: strictly speaking, Python doesn't support them. However, you can pretend it does by using `tuple()`:

```
>>> tuple(student.gpa for student in students
...       if student.major == "Computer Science")
(3.6, 2.5)
```

This creates a tuple, *but it's not a tuple comprehension*. You're calling the `tuple` constructor, and passing it a single argument. What's that argument? A generator expression! In other words, you're doing this:

```
>>> cs_students = (
...     student.gpa for student in students
...     if student.major == "Computer Science"
...     )
>>> type(cs_students)
<class 'generator'>
>>> tuple(cs_students)
(3.6, 2.5)
>>>
>>> # Same as:
... tuple((student.gpa for student in students
...        if student.major == "Computer Science"))
(3.6, 2.5)
>>> # But you can omit the inner parentheses.
```

`tuple`'s constructor takes an iterator as an argument. The `cs_students` is a generator object (created by the generator expression), and a generator object is an iterator. So you can *pretend* like Python has tuple comprehensions, using `"tuple("` as the opener and `")"` as the close. In fact, this also gives you alternate ways to create dictionary and set comprehensions:

```
>>> # Same as:
... # { student.name: student.gpa for student in students }
>>> dict((student.name, student.gpa)
...      for student in students)
{'Jim Smith': 3.6, 'Penny Gilmore': 3.9,
 'Alisha Jones': 2.5, 'Ryan Spencer': 3.1,
 'Todd Reynolds': 3.4}
>>> # Same as:
```

```
... # { student.major for student in students }
>>> set(student.major for student in students)
{'Computer Science', 'Basket Weaving', 'Economics'}
```

Remember, when you pass a generator expression into a function, you can omit the inner parentheses. That's why you can, for example, type

```
tuple(f(x) for x in numbers)
```

instead of

```
tuple((f(x) for x in numbers))
```

One last point. Generator expressions are a scalable analogue of list comprehensions; is there any such equivalent for dicts or sets? No, it turns out. If you need to lazily generate key-value pairs or unique elements, your best bet is to write a generator function.

Limits of Comprehensions

Comprehensions have a few wrinkles people sometimes trip over. Consider the following code:

```
# Read in the lines of a file, stripping leading and
# trailing whitespace, skipping any empty or
# whitespace-only lines.
trimmed_lines = []
for line in open('wombat-story.txt'):
    line = line.strip()
    if line != "":
        trimmed_lines.append(line)

print("Got {} lines".format(len(trimmed_lines)))
```

Straightforward enough - we're building a list named `trimmed_lines`. The resulting list has all leading and trailing whitespace removed from its elements, skipping any empty lines (or lines that were just whitespace to begin with). It's not hard to imagine needing to do something like this in a real program.

Now… how would you do this using a list comprehension? Here's a first try:

```
with open('wombat-story.txt') as story:
    trimmed_lines = [
        line.strip()
        for line in story
        if line.strip() != ""
        ]

print("Got {} lines".format(len(trimmed_lines)))
```

This works. Notice, though, that `line.strip()` appears twice. That's wasting CPU cycles compared to the for-loop version, which only calls `line.strip()` once. Stripping whitespace from a string isn't *that* expensive, computationally speaking. But sooner or later, you will want to do something where this matters:

```
>>> values = [
...     expensive_function(n)
...     for n in range(BIG_NUMBER)
...     if expensive_function(n) > 0
...     ]
```

So how can you create this as a list comprehension, while calling `expensive_function` only once? It turns out there is no direct way to do this. There some perhaps-too-clever solutions, such as memoizing (which can easily overuse memory), nesting a generator expression inside (probably the best choice), or making an intermediate list (if it's small).

If the sequence you need fits the pattern above, you might consider building it the old-fashioned way, using a `for` loop or a generator function. Or if you still want to use a comprehension, use an intermediate generator expression. The result is fairly readable and understandable (at least for you, having read this far):

```
>>> intermediate_values = (
...     expensive_function(n)
...     for n in range(10000)
... )
>>>
>>> values = [
...     intermediate_value
```

```
...     for intermediate_value in intermediate_values
...     if intermediate_value > 0
...     ]
```

Another limitation is that comprehensions must be built on one element at a time. The best way to see this is to imagine a list composed of inlined key-value pairs - flattened, in other words, so even-indexed elements are keys, and each key's value comes right after it. Imagine a function that converts this to a dictionary:

```
>>> # Price per pound of fruits & vegetables, in dollars.
... prices_flat_list = [
...     "orange", 0.70, "banana", 0.86,
...     "cantaloupe", 0.63, "bok choy", 1.56,
...     "coconuts", 1.06 ]

>>> list2dict(prices_flat_list)
{'banana': 0.86, 'bok choy': 1.56, 'cantaloupe': 0.63, 'orange':
0.7, 'coconuts': 1.06}
```

Here's one way to implement `list2dict`:

```
# Converts a "flattened" list into an "unflattened" dict.
def list2dict(flattened):
    assert len(flattened) % 2 == 0,
        "Input must be list of key-value pairs"
    unflattened = dict()
    for i in range(0, len(flattened), 2):
        key, value = flattened[i], flattened[i+1]
        unflattened[key] = value
    return unflattened
```

Look at `list2dict`'s for loop. It runs through the even index numbers of elements in `flattened`, rather than the elements of `flattened` directly. This allows it to refer to two different list elements each time through the for

loop. But this turns out to be something which just can't be expressed in the semantics of a Python comprehension. Generally, a comprehension operates by looking at each element in some source sequence, one at a time; it can't peek at neighboring elements.

Another example: a function grouping the elements of a sequence by some criteria - for example, the first letter of a string:

```
>>> names = ["Joe", "Jim", "Todd",
...     "Tiffany", "Zelma", "Gerry", "Gina"]
>>> grouped_names = group_by_first_letter(names)
>>> grouped_names['j']
['Joe', 'Jim']
>>> grouped_names['z']
['Zelma']
```

Here's one way to implement the grouping function:

```
from collections import defaultdict
def group_by_first_letter(items):
    grouped = defaultdict(list)
    for item in items:
        key = item[0].lower()
        grouped[key].append(item)
    return grouped
```

Again, the semantics of Python comprehensions aren't built to support this kind of algorithm. In functional programming terms, comprehensions can use map and filter operations, but not reduce or fold. Fortunately, this covers many use cases. I point out these limitations to help you avoid wasting time trying to figure them out; in spite of them, I find comprehensions to be a valuable part of my daily Python toolbox, and I'm sure you will too. □

1 Technically, the condition doesn't have to depend on the variable, but it's hard to imagine building a useful list comprehension this way.

2 I like to think future version control tools will automatically resolve this kind of situation. I believe it will require the tool to have some knowledge of the language syntax, so it can parse and reason about different clauses in a line of code.

3 Refresher: The `range` built-in returns an iterator over a sequence of integers, and can be called with 1, 2, or 3 arguments. The most general form is `range(start, stop, step)`, beginning at `start`, going up to *but not including* `stop`, in increments of `step`. Called with 2 arguments, the step-size defaults to one; with one argument, that argument is the `stop`, and the sequence starts at zero. In Python 2, use `xrange` instead of `range`.

ADVANCED FUNCTIONS

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

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

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

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

Accepting & Passing Variable Arguments

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

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

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

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

If you call the function with no arguments, `args` is an empty tuple. Otherwise, it is a tuple composed of those arguments passed, in order. This is

different from declaring a function that takes a single argument, which happens to be of type list or tuple:

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

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

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

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

```
def read_files(*paths):
    data = ""
    for path in paths:
```

```
        with open(path) as handle:
            data += handle.read()
    return data
```

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

Argument Unpacking

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Variable Keyword Arguments

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

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

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

```
>>> def bar(x, y, z):
...     return x + y * z
...
```

```
>>> bar(z=2, y=3, x=4)
10
```

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

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

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

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

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

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

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

```
        if key not in config:
            config[key] = value
```

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

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

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

Keyword Unpacking

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

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

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

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

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

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

Combining Positional and Keyword Arguments

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

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

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

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

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

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

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

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

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

```
def combined4(x, *args): pass
def combined5(u, v, w, *args, **kwargs): pass
def combined6(*args, x, y): pass
```

Violating this order will cause errors:

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

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

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

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

```
>>> commodities_data = read_data_from_files(
...     "commodities.xml", "xml")
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
TypeError: read_data_from_files() missing 1 required keyword-only
argument: 'format'
```

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

One more example - a list of dictionary objects:

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

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

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

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

I don't know about you, but having nearly-identical functions like this drives me nuts. The way out is to realize the comparison is based on a value *derived* from the element - not the value of the element itself. In other words: each

cycle through the for loop, the two elements are **not** themselves compared. What is compared is some derived, calculated value: `int(item)`, or `abs(item)`, or `item["gpa"]`.

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

This lets us define a very generic max function:

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

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

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

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.

Key Functions in Python

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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`.

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

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

DECORATORS

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

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

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

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

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

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

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

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

The Basic Decorator

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here's the result:

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

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

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

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

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

Can you spot what went wrong?

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

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

This decorator is compatible with *any* Python function:

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

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

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

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

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

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

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

Simply defining `wrapper(*args, **kwargs)` causes `self` to be considered one of the `args`; such a decorator works just as well with both functions and methods. But if a wrapper is defined to require `self`, that

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

Now, here's a different decorator:

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

It could be applied like this:

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

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

Here's the output when you execute:

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

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

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

Data In Decorators

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

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

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

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

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

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

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

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

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

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

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

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

```
            func.__name__, data["total"] / data["count"]))
        return func(*args, **kwargs)
    return wrapper
```

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

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

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

And that produces a problem:

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

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

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

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

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

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

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

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

```
def running_average(func):
    data = {"total" : 0, "count" : 0}
    def wrapper(*args, **kwargs):
        val = func(*args, **kwargs)
        data["total"] += val
```

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

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

This is why `running_average` produces the correct behavior:

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

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

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

```
... foo(1)
Average of foo so far: 3.0
3
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

But that just creates a different problem:

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

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

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

This finally works correctly:

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

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

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

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

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

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

Data in Decorators for Python 2

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

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

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

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

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

```
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`.

Decorators That Take Arguments

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

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

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

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

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

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

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

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

something like this:

```
@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 â€œadd(2)â€• in this case. That expression is evaluated. This all happens *at compile time*. Evaluating the decorator expression means executing `add(2)`, which will return a function object. That function object is the decorator. It's named `decorator` inside the body of the `add` function, but it doesn't really have a name at the top level; it's just applied to `f`.

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

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

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

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

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

Remember that `@` is a shorthand:

```
# 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.

Class-based Decorators

I lied to you.

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

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

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

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

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

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

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

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

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

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

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

```
>>> @PrintLog
... def foo(x):
...     print(x + 2)
...
>>> @PrintLog
```

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

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

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

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

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

Another benefit is when you prefer to accumulate state in object attributes, instead of a closure. For example, the `countcalls` function decorator

above could be implemented as a class:

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

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

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

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

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

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

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

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

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

Decorators For Classes

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

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

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

```
class Penny:
    value = 1

penny = Penny()
```

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

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

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

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

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

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

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

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

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

This has the disadvantage of creating a new type:

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

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

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

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

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

Note the IDs are the same:

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

Preserving the Wrapped Function

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

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

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

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

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

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

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

Contrast this with the default behavior:

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

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

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

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

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

So what should you do in practice?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXCEPTIONS AND ERRORS

Errors happen. That's why every practical programming language provides a rich framework for dealing with them.

Python's error model is based on *exceptions*. Some of you reading this are familiar with exceptions, and some are not; some of you have used exceptions in other languages, and not yet with Python. This chapter is for all of you.

If you are familiar with how exceptions work in Java, C++ or C#, you'll find Python uses similar concepts, even if the syntax is completely different. And beyond those similarities lie uniquely Pythonic patterns.

We'll start with the basics some of you know. Even if you've used Python exceptions before, I recommend reading all of this chapter. Odds are you will learn useful things, even in sections which appear to discuss what you've seen before.

The Basic Idea

A n *exception* is a way to interrupt the normal flow of code. When an exception occurs, the block of Python code will stop executing - literally in the middle of the line - and immediately jump to *another* block of code, designed to handle the situation.

Often an exception means an error of some sort, but it doesn't have to be. It can be used to signal anticipated events, which are best handled in an interrupt-driven way. Let's illustrate the common, simple cases first, before exploring more sophisticated patterns.

You've already encountered exceptions, even if you didn't realize it. Here's a little program using a `dict`:

```
# favdessert.py
def describe_favorite(category):
    "Describe my favorite food in a category."
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    return "My favorite {} is {}.".format(
        category, favorites[category])

message = describe_favorite("dessert")
print(message)
```

When run, this program exits with an error:

```
Traceback (most recent call last):
  File "favdessert.py", line 12, in <module>
    message = describe_favorite("dessert")
```

```
  File "favdessert.py", line 10, in describe_favorite
    category, favorites[category])
KeyError: 'dessert'
```

When you look up a missing dictionary key like this, we say Python *raises* a *KeyError*. (In other languages, the terminology is "throw an exception". Same idea; Python uses the word "raise" instead of "throw".) That `KeyError` is an *exception*. In fact, most errors you see in Python are exceptions. This includes `IndexError` for lists, `TypeError` for incompatible types, `ValueError` for bad values, and so on. When an error occurs, Python responds by raising an exception.

An exception needs to be handled. If not, your program will crash. You handle it with *try-except* blocks. They look like this:

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

Notice the structure. You have the keyword `try`, followed by an indented block of code, immediately followed by `except KeyError`, which has its own block of code. We say the `except` block *catches* the `KeyError` exception.

Run the program with these new lines, and you get the following output:

```
I have no favorite dessert. I love them all!
```

Importantly, the new program exits successfully; its exit code to the operating system indicates "success" rather than "failure".

Here's how `try` and `except` work:

- Python starts executing lines of code in the `try:` block.
- If Python gets to the end of the `try` block and no exceptions are raised, Python skips over the `except` block completely. None of its lines are executed, and Python proceeds to the next line after (if there is one).
- If an exception is raised anywhere in the `try` block, the program immediately stops - literally in the middle of the line; no further lines in the `try` block will be executed. Python then checks whether the exception type (`KeyError`, in this case) matches an `except` clause. If so, it jumps to the matching block's first line.
- If the exception does *not* match the `except` block, the exception ignores it, acting like the block isn't even there. If no higher-level code has an `except` block to catch it, the program will crash.

Let's wrap these lines of code in a function:

```
def print_description(category):
    try:
        message = describe_favorite(category)
        print(message)
    except KeyError:
        print("I have no favorite {}. I love them
all!".format(category))
```

Notice how `print_description` behaves differently, depending on what you feed it:

```
>>> print_description("dessert")
I have no favorite dessert. I love them all!
>>> print_description("appetizer")
My favorite appetizer is calamari.
>>> print_description("beverage")
My favorite beverage is coffee.
>>> print_description("soup")
I have no favorite soup. I love them all!
```

Exceptions aren't just for damage control. You will sometimes use them as a flow-control tool, to deal with ordinary variations you know can occur at runtime. Suppose, for example, your program loads data from a file, in JSON format. You import the `json.load` function in your code:

```
from json import load
```

`json` is part of Python's standard library, so it's always available. Now, imagine there's an open-source library called `speedyjson`,[1] with a `load` function just like what's in the standard library - except twice as fast. And your program works with BIG json files, so you want to preferentially use the `speedyjson` version when available. In Python, importing something that doesn't exist raises an `ImportError`:

```
# If speedyjson isn't installed...
>>> from speedyjson import load
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ImportError: No module named 'speedyjson'
```

How can you use `speedyjson` if it's there, yet gracefully fall back on `json` when it's not? Use a try-except block:

```
try:
    from speedyjson import load
except ImportError:
    from json import load
```

If `speedyjson` is installed and importable, `load` will refer to its version of the function in your code. Otherwise you get `json.load`.

A single `try` can have multiple `except` blocks. For example, `int()` will raise a `TypeError` if passed a nonsensical type; it raises `ValueError` if the type is acceptable, but its value can't be converted to an integer.

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

More realistically, you might log different error events[2] with different levels of severity:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

If an exception is raised, Python will check whether its type matches the first `except` block. If not, it checks the next. The first matching `except` block is executed, and all others are skipped over entirely - so you will never have more than one of the `except` blocks executed for a given `try`. Of course, if none of them match, the exception continues rising until something catches it. (Or the process dies.)

There's a good rule of thumb, which I suggest you start building as a habit now: *put as little code as possible in the `try` block*. You do this so your `except` block(s) will not catch or mask errors they should not.

Sometimes you will want to have clean-up code that runs *no matter what*, even if an exception is raised. You can do this by adding a `finally` block:

```
try:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

The code in the `finally` block is *always* executed. If an exception is raised in the `try` block, Python will immediately jump to the `finally` block, run its lines, then raise the exception. If an exception is not raised, Python will run all the lines in the `try` block, then run the lines in the `finally` block. It's a way to say, "run these lines no matter what".

You can also have one (or more) except clauses:

```
try:
    line1
    line2
    # etc.
except FirstException:
    line1
    line2
    # etc.
except SecondException:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

What's executed and when depends on whether an exception is raised. If not, the lines in the `try` block run, followed by the lines in the `finally` block; none of the `except` blocks run. If an exception *is* raised, and it matches one of the `except` blocks, then the `finally` block runs *last*. The order is: the `try` block (up until the exception is raised), then the matching `except` block, and then the finally block.

What if an exception is raised, but there is no matching except block? The `except` blocks are ignored, because none of them match. The lines of code in `try` are executed, up until the exception is raised. Python immediately jumps to the `finally` block; when its lines finish, only then is the exception raised.

It's important to understand this ordering. When you include a `finally` block, and an exception is raised, the code in the `finally` block interjects

itself between the code that could run in the `try` block, and the raising of the exception. A `finally` block is like insurance, for code which *must* run, no matter what.

Here's a good example. Imagine writing control code that does batch calculations on a fleet of cloud virtual machines. You issue an API call to rent them, and pay by the hour until you release them. Your code might look something like:

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Now imagine `running_job.wait()` raises a `socket.timeout` exception (which means the network connection has timed out). This causes a stack trace, and the program crashes, or maybe some higher-level code actually catches the exception.

Regardless, now `fleet.terminate()` is never called. Whoops. That could be *really* expensive.

To save your bank balance (or keep your job), rewrite the code using a `finally` block:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
finally:
    fleet.terminate()
```

This code expresses the idea: "no matter what, terminate the fleet of rented virtual machines." Even if an error in `fleet.submit_job(job)` or `running_job.wait()` makes the program crash, it calls `fleet.terminate()` with its dying breath.

Let's look at dictionaries again. When working directly with a dictionary, you can use the "if key in dictionary" pattern to avoid a `KeyError`, instead of try/except blocks:

```
# Another approach we could have taken with favdessert.py
def describe_favorite_or_default(category):
    'Describe my favorite food in a category.'
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    if category in favorites:
        message = "My favorite {} is {}.".format(
        category, favorites[category])
    else:
        message = "I have no favorite {}. I love them
all!".format(category)
    return message
```

```
message = describe_favorite_or_default("dessert")
print(message)
```

The general pattern is:

```
# Using "if key in dictionary" idiom.
if key in mydict:
    value = mydict[key]
else:
    value = default_value

# Contrast with "try/except KeyError".
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Many developers prefer using the "if key in dictionary" idiom, or using `dict.get()`. But these aren't always the best choice. They are only options if your code has direct access to the dictionary, for one thing. Maybe `describe_favorite()` is part of a library, and you can't change it. Even if it's open-source, you have better things to do than fork a library every time a function interface isn't convenient. Or maybe `describe_favorite()` is code you control, but you just don't *want* to change it, for any number of good reasons. A try-except block catching `KeyError` solves all these problems, because it lets you handle the situation without modifying any code inside `describe_favorite()` itself.

Exceptions Are Objects

An exception is an object: an instance of an exception class. `KeyError`, `IndexError`, `TypeError` and `ValueError` are all built-in classes, which inherit from a base class called `Exception`. Writing code like `except KeyError:` means "if the exception just raised is of type `KeyError`, run this block of code."

So far, we haven't dealt with those exception objects directly. And often, you don't need to. But sometimes you want more information about what happened, and capturing the exception object can help. Here's the structure:

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

where *ExceptionClass* is some exception class, like `KeyError`, etc. In the except block, `exception_object` will be an instance of that class. You can choose any name for that variable; no one actually calls it `exception_object`, preferring shorter names like `ex`, `exc`, or `err`. The methods and contents of that object will depend on the kind of exception, but almost all will have an attribute called `args`. That will be a tuple of what was passed to the exception's constructor. The `args` of a `KeyError`, for example, will have one element - the missing key:

```
# Atomic numbers of noble gasses.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(elements):
```

```
    for element in elements:
        print('Atomic number of {} is {}'.format(
              element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print('Missing data for element: ' + missing_element)
```

Running this code gives you the following output:

```
Atomic number of Ne is 10
Atomic number of Ar is 18
Missing data for element: Br
```

The interesting bit is in the `except` block. Writing `except KeyError as err` stores the exception object in the `err` variable. That lets us look up the offending key, by peeking in `err.args`. Notice we could not get the offending key any other way, unless we want to modify `show_element_info` (which we may not want to do, or perhaps *can't* do, as described before).

Let's walk through a more sophisticated example. In the `os` module, the `makedirs` function will create a directory:

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("riddles")
```

By default, if the directory already exists, `makedirs` will raise `FileExistsError`:[3] Imagine you are writing a web application, and need to create an upload directory for each new user. That directory should

not exist; if it does, that's an error and needs to be logged. Our upload-directory-creating function might look like this:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

It's great we are detecting and logging the error, but the error message isn't informative enough to be helpful. We at least need to know the offending username, but it's even better to know the directory's full path (so you don't have to dig in the code to remind yourself what `UPLOAD_ROOT` was set to).

Fortunately, `FileExistsError` objects have an attribute called `filename`. This is a string, and the path to the already-existing directory. We can use that to improve the log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

Only the `except` block is different. That `filename` attribute is perfect for a useful log message.

Raising Exceptions

`ValueError` is a built-in exception that signals some data is of the correct type, but its format isn't valid. It shows up everywhere:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a
number'
```

Your own code can raise exceptions, just like `int()` does. You should, in fact, so you have better error messages. (And sometimes for other reasons - more on that later.) You do so with the *raise* statement. The most common form is this:

```
raise ExceptionClass(arguments)
```

For `ValueError` specifically, it might look like:

```
def positive_int(value):
    number = int(value)
    if number <= 0:
        raise ValueError("Bad value: " + str(value))
    return number
```

Focus on the `raise` line in `positive_int`. You simply create an instance o f `ValueError`, and pass it directly to `raise`. Really, the syntax is `raise exception_object` - though usually you just create the object inline. `ValueError`'s constructor takes one argument, a descriptive string. This shows up in stack traces and log messages, so be sure to make it informative and useful:

```
>>> positive_int("-3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -3
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -7.0
```

Let's show a more complex example. Imagine you have a `Money` class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        'Renders the object nicely on the prompt.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods, which aren't important to us now.
```

Your code needs to create `Money` objects from string values, like "$140.75". The constructor takes dollars and cents, so you create a function to parse that string and instantiate `Money` for you:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This function[4] works like this:

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

This error isn't clear; you must read the source and think about it to understand what went wrong. We have better things to do than decrypt stack traces. You can improve this function's usability by having it raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

The error message is now much more informative:

```
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: 'Big money'
```

Catching And Re-raising

In an `except` block, you can re-raise the current exception. It's very simple; just write `raise` by itself, with no arguments:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

Notice you don't need to store the exception object in a variable. It's a shorthand, exactly equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

This "catch and release" only works in an `except` block. It requires that some higher-level code will catch the exception and deal with it. Yet it enables several useful code patterns. One is when you want to delegate handling the exception to higher-level code, but also want to inject some extra behavior closer to the exception source. For example:

```
try:
    process_user_input(value)
except ValueError:
    logging.info("Invalid user input: %s", value)
    raise
```

If `process_user_input` raises a `ValueError`, the `except` block will execute the logging line. Other than that, the exception propagates as normal.

It's also useful when you need to execute code before deciding whether to re-raise the exception at all. Earlier, we used a try/except block pair to create an upload directory, logging an error if it already exists:

```
# Remember this? Python 3 code, from earlier.
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

This relies on `FileExistsError`, which was introduced in Python 3. How could you do this in Python 2? Even if you no longer write code in Python 2, it's worth studying the different approach required, as it demonstrates a widely useful exception-handling pattern. Let's take a look.

`FileExistsError` subclasses the more general `OSError`. This exception type has been around since the early days of Python, and in Python 2, `makedirs` simply raises `OSError`. But `OSError` can indicate many problems other than the directory already existing: a lack of filesystem permissions, a system call getting interrupted, even a timeout over a network-mounted filesystem. We need a way to discriminate between these possibilities.

`OSError` objects have an `errno` attribute, indicating the precise error. These correspond to the variable `errno` in a C program, with different integer values meaning different error conditions. Most higher-level languages - including Python - reuse the constant names defined in the C API; in particular, the standard constant for "file already exists" is `EEXIST` (which happens to be set to the number 17 in most implementations). These constants are defined in the `errno` module in Python, so we just type `from errno import EEXIST` in our program.

In versions of Python with `FileExistsError`, the general pattern is:

- Optimistically create the directory, and
- if `FileExistsError` is raised, catch it and log the event.

In Python 2, we must do this instead:

- Optimistically create the directory.
- if `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

The code:

```
# How to accomplish the same in Python 2.
import os
import logging
```

```
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
        logging.error("Upload dir already exists: %s",
            err.filename)
```

The only difference between the Python 2 and 3 versions is the "except" clause. But there's a lot going on there. First, we're catching `OSError` rather than `FileExistsError`. But we may or may not re-raise the exception, depending on the value of its `errno` attribute. Basically, a value of `EEXIST` means the directory already exists. So we log it and move on. Any other value indicates an error we aren't prepared to handle right here, so re-raise in order to pass it to higher level code.

The Most Diabolical Python Anti-Pattern

You know about "design patterns": time-tested solutions to common code problems. And you've probably heard of "anti-patterns": solutions people often choose to a code problem, because it *seems* to be a good approach, but actually turn out to be harmful.

In Python, one antipattern is most harmful of all.

I wish I could not even tell you about it. If you don't know it exists, you can't use it in your code. Unfortunately, you might stumble on it somewhere and adopt it, not realizing the danger. So, it's my duty to warn you.

Hereâ€™s the punchline. The following is the most self-destructive code a Python developer can write:

```
try:
    do_something()
except:
    pass
```

Python lets you completely omit the argument to `except`. If you do that, it will catch *every exception*. That's pretty harmful right there; remember, the more pin-pointed your `except` clauses are, the more precise your error handling can be, without sweeping unrelated errors under the rug. And typing `except:` will sweep *every* unrelated error under the rug.

But it's much worse than that, because of the `pass` in the `except` clause. What `except: pass` does is silently and invisibly hide error conditions

that you'd otherwise quickly detect and fix.

(Instead of `"except:"`, you'll sometimes see variants like `"except Exception:"` or `"except Exception as ex:"`. They amount to the same thing.)

This creates the **worst kind of bug**. Have you ever been troubleshooting a bug, and just couldn't figure out where in the code base it came from, even after hours of searching, getting more and more frustrated as the minutes and hours roll by? *This is how you create that in Python.*

I first understood this anti-pattern after joining an engineering team, in an explosively-growing Silicon Valley start-up. The company's product was a web service, which needed to be up 24/7. So engineers took turns being "on call" in case of a critical issue. An obscure Unicode bug somehow kept triggering, waking up an engineer - in the middle of the night! - several times a week. But no one could figure out how to reproduce the bug, or even track down exactly how it was happening in the large code base.

After a few months of this nonsense, some of the senior engineers got fed up and devoted themselves to rooting out the problem. One senior engineer did nothing for *three full days* except investigate it, ignoring other responsibilities as they piled up. He made some progress, and took useful notes on what he found, but in the end did not figure it out. He ran out of time, and had to give up.

Then, a second senior engineer took over. Using those notes as a starting point, he also dug into it, ignoring emails and other commitments for *another*

three full days to track down the problem. And he failed. He make progress, adding usefully to the notes. But in the end, he had to give up too, when other responsibilities could no longer be ignored.

Finally, after these six days, they passed the torch to me - the new engineer on the team. I wasn't too familiar with the code base, but their notes gave me a lot to go on. So I dove in on Day 7, and completely ignored everything else for six hours straight.

And finally, late in the day, I was able to isolate the problem to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

That was it. The data in `location_data` was corrupted, causing the `extract_address` call to raise a `UnicodeError`. Which the program then *completely silenced*. Not even producing a stack trace; simply moving on, as if nothing had happened.

After nearly *seven full days* of engineer effort, we pinpointed the error to this one block of code. I un-suppressed the exception, and was almost immediately able to reproduce the bug - with a full and very informative stack trace.

Once I did that, can you guess how long it took us to fix the bug?

TEN MINUTES.

That's right. A full WEEK of engineer time was wasted, all because this anti-pattern somehow snuck into our code base. Had it not, then the first time it woke up an engineer, it would have been obvious what the problem was, and how to fix it. The code would have been patched by the end of the day, and we would all have moved on to bigger and better things.

The cruelty of this anti-pattern comes from how it completely hides *all* helpful information. Normally, when a bug causes a problem in your code, you can inspect the stack trace; identify what lines of code are involved; and start solving it. With The Most Diabolical Python Antipattern (TMDPA), none of that information is available. What line of code did the error come from? Which *file* in your Python application, for that matter? In fact, what was the exception type? Was it a `KeyError`? A `UnicodeError`? Or even a `NameError`, coming from a mis-typed variable name? Was it `OSError`, and if so, what was its `errno`? You don't know. You *can't* know.

In fact, TMDPA often **hides the fact that an error even occurs**. This is one of the ways bugs hide from you during development, then sneak into production, where they're free to cause real damage.

We never did figure out why the original developer wrote `except: pass` to begin with. I think that at the time, `location_data` may have sometimes been empty, causing `extract_address` to innocuously raise a `ValueError`. In other words, if `ValueError` was raised, it was appropriate to ignore that and move on. By the time the other two engineers and I were involved, the code base had changed so that was no longer how

things worked. But the broad `except` block remained, like a land mine lurking in a lush field.

So why do people do this? Well, no one *wants* to wreak such havoc in their Python code, of course. People do this because they expect errors to occur in the normal course of operation, in some specific way. They are simply catching too broadly, without realizing the full implications.

So what do you do instead? There are two basic choices. In most cases, it's best to modify the `except` clause to catch a more specific exception. For the situation above, this would have been a much better choice:

```
try:
    extract_address(location_data)
except ValueError:
    pass
```

Here, `ValueError` is caught and appropriately ignored. If `UnicodeError` raises, it propagates and (if not caught) the program crashes. That would have been *great* in our situation. The error log would have a full stack trace clearly telling us what happened, and we'd be able to fix it in ten minutes.

As a variation, you may want to insert some logging:

```
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

The other reason people write `except: pass` is a bit more valid. Sometimes, a code path simply must broadly catch all exceptions, and continue running regardless. This is common in the top-level loop for a long-running, persistent process. The problem is that `except: pass` hides all information about the problem, including that the problem even exists.

Fortunately, Python provides an easy way to capture that error event, and all the information you need to fix it. The `logging` module has a function called `exception`, which will log your message *along with the full stack trace of the current exception*. So you can write code like this:

```
import logging
def get_number():
    return int('foo')
try:
    x = get_number()
except:
    logging.exception('Caught an error')
```

The log will contain the error message, followed by a formatted stack trace spread across several lines:

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

This stack trace is *priceless*. Especially in more complex applications, it's often not enough to know the file and line number where an error occurs. It's at least as important to know *how* that function or method was called… what

path of executed code led to it being invoked. Otherwise you can never determine what conditions lead to that function or method being called in the first place. The stack trace, in contrast, gives you everything you need to know.

I wish "`except: pass`" was not valid Python syntax. I think much grief would be spared if it was. But it's not my call, and changing it now is probably not practical. Your only defense is to be vigilant. That includes educating your fellow developers. Does your team hold regular engineering meetings? Ask for five minutes at the next one to explain this antipattern, the cost it has to everyone's productivity, and the simple solutions.

Even better: if there are local Python or technical meetups in your area, volunteer to give a short talk - five to fifteen minutes. These meetups almost always need speakers, and you will be helping so many of your fellow developers in the audience.

There is a longer article explaining this situation at https://powerfulpython.com/blog/the-most-diabolical-python-antipattern/ . Simply sharing the URL will educate people too. And feel free to write your own blog post, with your own explanation of the situation, and how to fix it. Serve your fellow engineers by evangelizing this important knowledge.

1 Not a real library, so far as I know. But after this book is published, I'm sure one of you will make a library with that name, just to mess with me.

2 Especially in larger applications, exception handling often integrates with logging. See the logging chapter for details.

3 Python 2 does something different, and more complex. We'll cover that in detail later. For now, keep reading.

4 It's better to make this a class method of `Money`, rather than a separate function. That is a separate topic, though; see `@classmethod` in the object-oriented patterns chapter for details.

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.

Quick Note on Python 2

This chapter uses Python 3 syntax. Python 2.7 is nearly the same, and I'll point out the few 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`.

Properties

In object-oriented programming, a *property* is a special sort of object attribute. 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, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    @property
    def fullname(self):
        return self.firstname + " " + self.lastname
```

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

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

Notice carefully the members here: there are two attributes called `firstname` and `lastname`, set in the constructor. There is also a method called `fullname`. But after creating the object, we reference `joe.fullname` as an attribute; we don't call `joe.fullname()` 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.fullname()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

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

```
>>> joe.fullname = "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 `fullname` to be writable, here is how you define the setter:

```
class Person:
    def __init__(self, firstname, lastname):
        self.firstname = firstname
        self.lastname = lastname

    @property
    def fullname(self):
        return self.firstname + " " + self.lastname

    @fullname.setter
    def fullname(self, value):
        self.firstname, self.lastname = value.split(" ", 1)
```

This lets us assign to `joe.fullname`:

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

```
>>> joe.lastname
'Smith'
>>> joe.fullname = "Joseph Smith"
>>> joe.firstname
'Joseph'
>>> joe.lastname
'Smith'
```

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

The code is actually correct, and designed to work this way. The `@property def fullname` 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 `fullname` exists *in the namespace of the class*, and it has a method named `fullname.setter`. This `fullname.setter` is a decorator that is applied to the next `def fullname`, christening it as the setter for the `fullname` 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, you can handle the `del` operation for the object attribute by decorating with `@fullname.deleter`. You won't need this very often, 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. You'll see a situation where that's extremely useful later.

Properties enable a useful collection of design patterns. One - as mentioned - is in creating read-only member variables. In `Person`, the `fullname` "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:

```
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.[1] 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. Suppose my event-management application has a `Ticket` class, representing tickets sold to concert-goers:

```
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 how 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?

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

```
# 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 methods need to violate this rule; you simply reference `self._price` and move on. But avoid that where you can, and you will tend to benefit from higher quality code.

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
    # And for 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. Other developers use it 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.

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. The 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.

Alternative Constructors: The Simple Factory

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

```
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, 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)
    if match is None:
        raise ValueError("Invalid amount: " + repr(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.** The final line returns an instance of `cls`, using its regular constructor. And `cls` refers to whatever the current class is: `Money`, `TipMoney`, or some other subclass.

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

```
def from_string(cls, amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    if match is None:
        raise ValueError("Invalid amount: " + repr(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:

```
import re
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):
        match = re.search(
            r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
        if match is None:
            raise ValueError("Invalid amount: " + repr(amount))
        dollars = int(match.group('dollars'))
        cents = int(match.group('cents'))
        return cls(dollars, cents)
```

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__`.

When using a new class, most developer's intuition will be to reach for the default constructor first, without thinking to check the provided class methods - if they even know about that feature of Python in the first place. So in that situation, you may need to educate your teammates. (Hint: Good examples in the class's code docs go a long way.)

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 already 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.

The Observer Pattern

The Observer pattern provides a "one to many" relationship. That's vague, so let's make it more specific.

In the observer pattern, there's one root object, called the *observable*. This object knows how to detect some kind of event of interest. It can literally be anything: a customer makes a new purchase; someone subscribes to an email list; or maybe it monitors a fleet of cloud instances, detecting when a machine's disk usage exceeds 75%. You use this pattern when the code to *reliably* detect the event of interest is at least slightly complicated; that detection code is encapsulated inside the observable.

Now, you also have other objects, called *observers*, which need to know when that event occurs, taking some action in response. You don't want to re-implement the robust detection algorithm in each, of course. Instead, these observers register themselves with the observable. The observable then notifies each observer - by calling a method on that observer - for each event. This separation of concerns is the heart of the observer pattern.

Now, I must tell you, I don't like the names of things in this pattern. The words "observable" and "observer" are a bit obscure, and sound confusingly similar - especially to those whose native tongue is not English. There is another terminology, however, which many developers find easier: *pub-sub*.

In this formulation, instead of "observable", you create a *publisher* object, which watches for events. And you have one or more *subscribers* who ask that publisher to notify them when the event happens. I've found the pattern

is easier to reason about when looked at in this way, so that's the terminology I'm going to use.

Let's make this concrete, with code.

The Simple Observer

We'll start with the basic observer pattern, as it's often documented in design pattern books - except we'll translate it to Python. In this simple form, each subscriber must implement a method called `update`. Here's an example:

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

`update` takes a string. It's okay to define an `update` method taking other arguments, or even calling it something other than `update`; the publisher and subscriber just need to agree on the protocol. But we'll use a string.

Now, when a publisher detects an event, it notifies the subscriber by calling its `update` method. Here's what a basic `Publisher` class looks like:

```
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)
    # Plus other methods, for detecting the event.
```

Let's step through:

- A publisher needs to keep track of its subscribers, right? We'll store them in a set object, named `self.subscribers`, created in the constructor.
- A subscriber is added with `register`. Its argument `who` is an instance of `Subscriber`. Who calls `register`? It could be anyone. The subscriber can register itself; or some external code can register a subscriber with a specific publisher.
- `unregister` is there in case a subscriber no longer needs to be notified of the events.
- When the event of interest occurs, the publisher notifies its subscribers by calling its `dispatch` method. Usually this will be invoked by the publisher itself, in some other method of the class (not shown) that implements the event-detection logic. It simply cycles through the subscribers, calling `.update()` on each.

Using these two classes in code is straightforward enough:

```
# Create a publisher and some subscribers.
pub = Publisher()
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Register the subscribers, so they get notified.
```

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

Now, the publisher can dispatch messages:

```
# Send a message...
pub.dispatch("It's lunchtime!")
# John unsubscribes...
pub.unregister(john)
# ... and a new message is sent.
pub.dispatch("Time for dinner")
```

Here's the output from running the above:

```
John got message "It's lunchtime!"
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

This is the basic observer pattern, and pretty close to how you'd implement the idea in languages like Java, C#, and C++. But Python's feature set differs from those languages. That means we can do different things.

So let's explore that. If we leverage Pythonic features, what does that give us?

A Pythonic Refinement

Python's functions are first-class objects. That means you can store a function in a variable - not the value returned when you call a function, but store the function itself - as well as pass it as an argument to other functions and methods. Some languages support this too (or something like it, such as

function pointers), but Python's strong support gives us a convenient opportunity for this design pattern.

The standard observer pattern requires the publisher hard-code a certain method - usually named `update` - that the subscriber must implement. But maybe you need to register a subscriber which doesn't have that method. What then? If it's your own class, you can probably just add it. Or if you are importing the subscriber class from another library (which you can't or don't want to modify), perhaps you can add the method by subclassing it.

Or sometimes you can't do any of those things. Or you *could*, but it's a lot of trouble, and you want to avoid it. What then?

Let's extend the traditional observer pattern, and make `register` more flexible. Suppose you have these subscribers:

```
# This subscriber uses the standard "update"
class SubscriberOne:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(
            self.name, message))
# This one wants to use "receive"
class SubscriberTwo:
    def __init__(self, name):
        self.name = name
    def receive(self, message):
        print('{} got message "{}"'.format(
            self.name, message))
```

`SubscriberOne` is the same subscriber class we saw before. `SubscriberTwo` is almost the same: instead of `update`, it has a method

named `receive`. Okay, let's modify `Publisher` so it can work with objects of either subscriber type:

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

There's a lot going on here, so let's break it down. Look first at the constructor: it creates a `dict` instead of a set. You'll see why in a moment.

Now focus on `register`:

```
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
        self.subscribers[who] = callback
```

It can be called with one or *two* arguments. With one argument, `who` is a subscriber of some sort, and `callback` defaults to `None`. Inside, `callback` is set to `who.update`. Notice the lack of parentheses; `who.update` is a *method object*. It's just like a function object, except it happens to be tied to an instance. And just like a function object, you can store it in a variable, pass it as an argument to another function, and so on.[2] So we're storing it in a variable called `callback`.

What if `register` is called with 2 arguments? Here's how that might look:

```
pub = Publisher()
alice = SubscriberTwo('Alice')
pub.register(alice, alice.receive)
```

`alice.receive` is another method object; inside, this object is assigned to `callback`. Regardless of whether `register` is called with one argument or two, the last line inserts `callback` into the dictionary:

```
        self.subscribers[who] = callback
```

Take a moment to appreciate the remarkable flexibility of Python dictionaries. Here, you are using an arbitrary instance of either `SubscriberOne` or `SubscriberTwo` as a key. These two classes are unrelated by inheritance, so from Python's viewpoint they are completely distinct types. And for that key, you insert a *method object* as its value. Python does this seamlessly, without complaint! Many languages would make you jump through hoops to accomplish this.

Anyway, now it's clear why `self.subscribers` is a `dict` and not a `set`. Earlier, we only needed to keep track to who the subscribers were. Now, we also need to remember the callback for each subscriber. These are used in the `dispatch` method:

```
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

`dispatch` only needs to cycle through the values,[3] because it just needs to call each subscriber's update method (even if it's not called `update`). Notice

we don't have to reference the subscriber object to call that method; the method object internally has a reference to its instance (i.e. its â€œselfâ€), so `callback(message)` calls the right method on the right object. In fact, the only reason we keep track of subscribers at all is so we can `unregister` them.

Let's put this together with a few subscribers:

```
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")
```

Here's the output:

```
Bob got message "It's lunchtime!"
Alice got message "It's lunchtime!"
John got message "It's lunchtime!"
Bob got message "Time for dinner"
Alice got message "Time for dinner"
```

Now, pop quiz. Look at the `Publisher` class again:

```
class Publisher:
    def __init__(self):
        self.subscribers = dict()
    def register(self, who, callback=None):
        if callback is None:
            callback = who.update
```

```
        self.subscribers[who] = callback
    def dispatch(self, message):
        for callback in self.subscribers.values():
            callback(message)
```

Here's the question: does `callback` have to be a method of the subscriber? Or can it be a method of a different object, or something else? Think about this before you continue…

It turns out `callback` can be *any callable*, provided it has a signature compatible with how it's called in `dispatch`. That means it can be a method of some other object, or even a normal function. This lets you register subscriber objects without an update method at all:

```
# This subscriber doesn't have ANY suitable method!
class SubscriberThree:
    def __init__(self, name):
        self.name = name
# ... but we can define a function...
todd = SubscriberThree('Todd')
def todd_callback(message):
    print('Todd got message "{}"'.format(message))
# ... and pass it to register:
pub.register(todd, todd_callback)
# And then, dispatch a message:
pub.dispatch("Breakfast is Ready")
```

Sure enough, this works:

```
Todd got message "Breakfast is Ready"
```

Several Channels

So far, we've assumed the publisher watches for only one kind of event. But what if there are several? Can we create a publisher that knows how to detect

all of them, and let subscribers decide which they want to know about?

To implement this, let's say a publisher has several *channels* that subscribers can subscribe to. Each channel notifies for a different event type. For example, if your program monitors a cluster of virtual machines, one channel signals when a certain machine's disk usage exceeds 75% (a warning sign, but not an immediate emergency); and another signals when disk usage goes over 90% (much more serious, and may begin to impact performance on that VM). Some subscribers will want to know when the 75% threshold is crossed; some, the 90% threshold; and some might want to be alerted for both. What's a good way to express this in Python code?

Let's work with the mealtime-announcement code above. Rather than jumping right into the code, let's mock up the *interface* first. We want to create a publisher with two channels, like so:

```
# Two channels, named "lunch" and "dinner".
pub = Publisher(['lunch', 'dinner'])
```

So the constructor is different; it takes a list of channel names. Let's also pass the channel name to `register`, since each subscriber will register for one or more:

```
# Three subscribers, of the original type.
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')

# Two args: channel name & subscriber
pub.register("lunch", bob)
pub.register("dinner", alice)
pub.register("lunch", john)
pub.register("dinner", john)
```

Now, on dispatch, the publisher needs to specify the event type. So just like with `register`, we'll prepend a channel argument:

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

When correctly working, we'd expect this output:

```
Bob got message "It's lunchtime!"
John got message "It's lunchtime!"
Alice got message "Dinner is served"
John got message "Dinner is served"
```

Pop quiz (and if it's practical, pause here to write Python code): how would you implement this new, multi-channel `Publisher`?

There are several approaches, but the simplest I've found relies on creating a separate `subscribers` dictionary for each channel. One approach:

```
class Publisher:
    def __init__(self, channels):
        # Create an empty subscribers dict
        # for every channel
        self.channels = { channel : dict()
                          for channel in channels }
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        subscribers = self.channels[channel]
        subscribers[who] = callback
    def dispatch(self, channel, message):
        subscribers = self.channels[channel]
        for subscriber, callback in subscribers.items():
            callback(message)
```

This `Publisher` has a dict called `self.channels`, which maps channel names (strings) to subscriber dictionaries. `register` and `dispatch` are not too different: they simply have an extra step, in which `subscribers` is looked up in `self.channels`. I use that variable just for readability, and I think it's well worth the extra line of code:

```
# Works the same. But a bit less readable.
    def register(self, channel, who, callback=None):
        if callback is None:
            callback = who.update
        self.channels[channel][who] = callback
```

These are some variations of the general observer pattern, and I'm sure you can imagine more. What I want you to notice are the options available in Python when you leverage function objects, and other Pythonic features.

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 270Âº and 270Âº, it evaluates to 180Âº instead of 540Âº.
  - If we subtract 180Âº from 90Âº, it evaluates to 270Âº instead of -90Âº.
  - 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 it's better if we can reuse the familiar arithmetic operators everyone knows. 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 standard operators. That includes arithmetic (+ - * / //), equality (==), inequality (!=), comparisons (< > >= <=), bit-shifting operations, and even concepts like exponentiation and absolute value.

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.

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. These are the magic methods.

When you discuss magic methods in face-to-face, verbal conversation, you'll find yourself saying things like "underscore underscore add underscore underscore" over and over. That's a lot of syllables, and you'll get tired of it fast. So people in the Python community use a kind of verbal abbreviation, with a word they invented: "dunder". That's not a real word; Python people made it up. When you say "dunder foo", it means "underscore underscore foo underscore underscore". This isn't used in writing, because it's not needed - you can just write `__foo__`. But at Python gatherings, you'll sometimes hear people say it. Use it; it saves you a lot of energy when talking.

Anyway, back to dunder add - I mean, `__add__`. For operations like addition - which take two values, and return a third - you write the method like this:

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

The first argument needs to be called "self", because this is Python. The second does not have to be called "other", but often is. 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 so on:

`__add__`	a + b
`__sub__`	a - b

`__mul__`	a * b
`__truediv__`	a / b (floating-point division)
`__mod__`	a % b
`__pow__`	a ** b

Essentially, Python translates `a + b` to `a.__add__(b)`; `a % b` to `a.__mod__(b)`; and so on. You can also hook into bit-operation operators:

`__lshift__`	a << b
`__rshift__`	a >> b
`__and__`	a & b
`__xor__`	a ^ b
`__or__`	a | b

So `a & b` translates to `a.__and__(b)`, for example. Since `__and__` corresponds to the bitwise-and operator (for expressions like `"foo & bar"`), you might wonder what the magic method is for *logical*-and (`"foo and bar"`), or logical-or (`"foo or bar"`). Sadly, there is none. For this reason, sometimes libraries will hijack the `&` and `|` operators to mean logical and/or instead of bitwise and/or, if the author feels the logical version is more important.

The default representation of an `Angle` object isn't very useful:

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

It tells us the type, and the hex object ID, but we'd rather it tell us something about 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))
30 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 a human-friendly representation. Imagine you have several subclasses (e.g., `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__`:

```
    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:

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

The official guideline is that the output of `__repr__` is something that can be passed to `eval()` to recreate the object exactly. It's not enforced by the language, and isn't always practical, or even possible. But when it is, doing so is useful for logging and debugging.

We also want to be able to compare two `Angle` objects. The most basic comparison is equality, provided by `__eq__`. It should return True or False:

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

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

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

By default, the == operator for objects is based off the object ID. That's rarely 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.

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__` (and `__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.

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.

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, 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 deep inside some nested loop. Starting with `total_ordering` takes little effort, and you can always simply remove it and hand-code the other magic methods if you need to.

Python 2 != Python 3

Some magic methods operate a bit differently in Python 2. For example,, 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)
```

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`. Watch out for this race condition if you employ magic methods in Python 2.

Rebelliously Misusing Magic Methods

Magic methods are interesting enough, and quite handy when you need them. A realistic currency type is a good example. But depending on the kind of applications you work on, it's not all that often you will need to define a class whose instances can be added, subtracted, or compared.

Things get much more interesting, though, when you don't follow the rules.

Here's a fascinating fact: methods like `__add__` are *supposed* to do addition. But it turns out Python doesn't require it. And methods like `__gt__` are *supposed* to return `True` or `False`. But if you write a `__gt__` which returns something that isn't a `bool`… Python won't complain at all.

This creates some *amazing* possibilities.

To illustrate, let me tell you about Pandas. As you may know, this is an excellent data-processing library. It's become extremely popular among data scientists who use Python (like some of you reading this). Pandas has a convenient data type called a `DataFrame`. It represents a two-dimensional collection of data, organized into rows, with labeled columns:

```
import pandas
df = pandas.DataFrame({
        'A': [-137, 22, -3, 4, 5],
        'B': [10, 11, 121, 13, 14],
        'C': [3, 6, 91, 12, 15],
    })
```

There are several ways to create a `DataFrame`; here I've chosen to use a dictionary. The keys are column names; the values are lists, which become that column's data. So you visually rotate each list 90 degrees:

```
>>> print(df)
     A    B   C
0 -137   10   3
1   22   11   6
2   -3  121  91
3    4   13  12
4    5   14  15
```

The rows are numbered for you, and the columns nicely labeled in a header. The `A` column, for example, has different positive and negative numbers.

Now, one of the many useful things you can do with a `DataFrame` is filter out rows meeting certain criteria. This doesn't change the original dataframe; instead, it creates a *new* dataframe, containing just the rows you want. For example, you can say "give me the rows of `df` in which the `A` column has a positive value":

```
>>> positive_a = df[df.A > 0]
>>> print(positive_a)
    A   B   C
1  22  11   6
3   4  13  12
4   5  14  15
```

All you have to do is pass in `"df > 0"` in the square brackets.

But there's something weird going on here. Take a look at the line in which `positive_a` is defined. Do you notice anything unusual there? Anything strange?

Here's what is odd: the expression `"df > 0"` ought to evaluate to either `True`, or `False`. Right? It's supposed to be a boolean value… with exactly *one bit* of information. But the source dataframe, `df`, has many rows. Realistic dataframes can easily have tens of thousands, even *millions* of rows of data. There's no way a boolean literal can express which of those rows to keep, and which to discard. **How does this even work?**

Well… turns out, it's not boolean at all:

```
>>> comparison = (df.A > 0)
>>> type(comparison)
<class 'pandas.core.series.Series'>
>>> print(comparison)
0    False
1     True
2    False
3     True
4     True
Name: A, dtype: bool
```

Yes, you can do that, thanks to Python's dynamic type system. Python translates `"df.A > 0"` into `"df.A.__gt__(0)"`. And that `__gt__` method doesn't have to return a bool. In fact, in Pandas, it returns a `Series` object (which is like a vector of data), containing `True` or `False` for each row. And when that's passed into `df[]` - the square brackets being handled by the `__getitem__` method - that `Series` object is used to filter rows.

To see what this looks like, let's re-invent part of the interface of Pandas. I'll create a library called `fakepandas`, which instead of `DataFrame` has a type called `Dataset`:

```
class Dataset:
    def __init__(self, data):
        self.data = data
        self.labels = sorted(data.keys())
    def __getattr__(self, label: str):
        "Makes references like df.A work."
        return Column(label)
    # Plus some other methods.
```

If I have a `Dataset` object named `ds`, with a column named `A`, the `__getattr__` method makes references like `ds.A` return a `Column` object:

```
import operator
class Column:
    def __init__(self, name):
        self.name = name
    def __gt__(self, value):
         return Comparison(self.name, value, operator.gt)
```

This `Column` class has a `__gt__` method, which makes expressions like `"ds.A > 0"` return an instance of a class called `Comparison`. It represents a lazy calculation, for when the actual filtering happens later. Notice its constructor arguments: a column name, a threshold value, and a callable to implement the comparison. (The `operator` module has a function called `gt`, taking two arguments, expressing a greater-than comparison).

You can even support complex filtering criteria like `ds[ds.C + 2 < ds.B]`. It's all possible by leveraging magic methods in these unorthodox ways. If you care about the details, there's an article delving into that.[4] My

goal here isn't to tell you how to re-invent the Pandas interface, so much as to get you to realize what's possible.

Have you ever implemented a compiler? If so, you know the parsing phase is a significant development challenge. Using Python magic methods in this manner does much of the hard work of lexing and parsing for you. And the best part is how natural and intuitive the result can be for end users. You are essentially implementing a mini-language on top of regular Python syntax, but consistently enough that people quickly become fluent and productive with its rules. And they often won't even think to ask why the rules seem to be bent; they won't notice `"df.A > 0"` isn't acting like a boolean. That's a clear sign of success. It means you designed your library so well, other developers become effortlessly productive.

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

2 This is all detailed in the "Advanced Functions" chapter.

3 In Python 2: Remember, use `.viewvalues()` instead of `.values()`.

4 See https://powerfulpython.com/blog/rebellious-magic-methods-python-syntax/ . The article explains these ideas in richer detail, and includes the full code of `fakepandas` and its unit test suite.

AUTOMATED TESTING AND TDD

Writing automated tests is one of those things that separates average developers from the best in the world. Master this skill, and you will be able to write *far* more complex and powerful software than you ever could before. It's a superpower, and changes the arc of your career.

There are roughly two kinds of readers for this chapter. Some of you have, so far, little or no experience writing automated tests, in any language. This chapter is primarily written for you. It introduces many fundamental ideas of test automation, explains the problems it is supposed to solve, and teaches how to apply Python's tools for doing so.

Or you might be someone with extensive experience using standard test frameworks in other languages: JUnit in Java, PHPUnit in PHP, and so on. Generally speaking, if you have mastered an xUnit framework in another language, and are fluent in Python, you may be able to start skimming the Python's `unittest` module docs[1] and be productive in minutes. Python's test library, `unittest`, was originally based on JUnit 3, and maps very closely to how most xUnit libraries work.[2]

If you are more experienced, I believe it’s worth your time to at least skim the chapter, and perhaps study it thoroughly. In writing, I invested a great deal of effort weaving in useful, real-world wisdom - both for software testing in general, and for Python specifically. This includes topics like how to organize Python test code; writing test code which is maintainable; useful, Python-only features like subtests; and even cognitive aspects of programming… getting into an enjoyable, highly productive "flow" state via test-driven development.

With that in mind, let’s start with the core ideas for writing automated tests. We’ll then focus on writing them for Python programs.

What is Test-Driven Development?

An *automated test* is a program that tests another program. Generally, it tests a specific portion of that program: a function, a method, a class, or some group of these things. We call that portion the "system under test". If the system under test is working correctly, the test passes; if not, our test catches that error, and immediately tells us what is wrong. Real applications accumulate many of these tests as development proceeds.

People have different names for different kinds of automated tests: unit tests, integration tests, end-to-end tests, etc. These distinctions can be useful, but we won't need to worry about them right now. They all share the same foundation.

In this chapter, we do *test-driven development*, or TDD. Test-driven development means you start working on each new feature or bugfix by writing the automated test for it **first**. You run that test, verify it fails, and only then do you write the actual code for the feature. You know you are done when the test passes.

This is a different process from implementing the feature first, **then** writing a test for it after. Writing the test first forces you to think through the interfaces of your code, answering the question "how will I know my code is working?" That immediate benefit is useful, but not the whole story.

The greatest mid-term benefits are mostly cognitive. As you become competent and comfortable with test-driven development, you learn to easily get into a state of flow - where you find yourself repeatedly implementing

feature after feature, keeping your focus with ease for long periods of time. You can honestly surprise and delight yourself with how much you've accomplished in a few hours of coding.

But the greatest benefits emerge over time. We've all done substantial refactorings of a large code base, changing fundamental aspects of its architecture.[3] Such refactorings - which threaten to break the application in confusing, hidden ways - become straightforward and safe using TDD. You take the existing body of tests, updating where needed and introducing new tests as appropriate. Then all you have to do is make them pass. It may still be a ton of work. But you can be fairly confident in the correctness and robustness of the result, instead of hoping and praying.

Among developers who know how to write tests, some love to do test-driven development in their day to day work. Some like to do it part of the time; some hate it, and do it rarely, or never. However, the absolute best way to quickly master unit testing is to strictly do test-driven development for a while. So I'll teach you how to do that. You don't have to do it forever if you don't want to.

Python's standard library ships with two modules for creating unit tests: `doctest` and `unittest`. Most engineering teams prefer `unittest`, as it is more full-featured than `doctest`. This isn't just a convenience. There is a real ceiling of complexity that `doctest` can handle, and real applications will quickly bump up against that limit. With `unittest`, the sky is more or less the limit.

In addition - as noted above - `unittest` maps almost exactly to the xUnit libraries used in many other languages. If you are already familiar with Python, and have used JUnit, PHPUnit, or any other xUnit library in any language, you will feel right at home with unittest. That said, `unittest` has some unique tools and idioms - partly because of differences in the Python language, and partly from unique extensions and improvements. We will learn the best of what `unittest` has to offer as we go along.

Unit Tests And Simple Assertions

Imagine a class representing an angle:

```
>>> small_angle = Angle(60)
>>> small_angle.degrees
60
>>> small_angle.is_acute()
True
>>> big_angle = Angle(320)
>>> big_angle.is_acute()
False
>>> funny_angle = Angle(1081)
>>> funny_angle.degrees
1
>>> total_angle = small_angle.add(big_angle)
>>> total_angle.degrees
20
```

As you can see, `Angle` keeps track of the angle size, wrapping around so it's in a range of 0 up to 360 degrees. There is also an `is_acute` method, to tell you if its size is under 90 degrees, and an `add` method for arithmetic.[4]

Suppose this `Angle` class is defined in a file named `angle.py`. Here's how we create a simple test for it - in a separate file, named `test_angle.py`:

```
import unittest
from angle import Angle

class TestAngle(unittest.TestCase):
    def test_degrees(self):
        small_angle = Angle(60)
        self.assertEqual(60, small_angle.degrees)
        self.assertTrue(small_angle.is_acute())
        big_angle = Angle(320)
```

```
        self.assertFalse(big_angle.is_acute())
        funny_angle = Angle(1081)
        self.assertEqual(1, funny_angle.degrees)

    def test_arithmetic(self):
        small_angle = Angle(60)
        big_angle = Angle(320)
        total_angle = small_angle.add(big_angle)
        self.assertEqual(20, total_angle.degrees,
                         'Adding angles with wrap-around')
```

As you look over this code, notice a few things:

- There's a class called `TestAngle`. You just define it, not create any instance of it. This subclasses `TestCase`.
- You define two methods, `test_degrees` and `test_arithmetic`.
- Both `test_degrees` and `test_arithmetic` have assertions, using some methods of `TestCase`: `assertEqual`, `assertTrue`, and `assertFalse`.
- The last assertion includes a custom message, as its third argument.

To see how this works, let's define a stub for the `Angle` class in `angles.py`:

```
# angle.py, version 1
class Angle:
    def __init__(self, degrees):
        self.degrees = 0
    def is_acute(self):
        return False
    def add(self, other_angle):
        return Angle(0)
```

This `Angle` class defines all the attributes and methods it is expected to have, but otherwise can't do anything useful. We need a stub like this to verify the test can run correctly, and alert us to the fact that the code isn't working yet.

The `unittest` module is not just used to define tests, but also to run them. You do so on the command line like this:

```
python3 -m unittest test_angles.py
```

When you run the test,[5] and you'll see the following output:

```
$ python3 -m unittest test_angle.py
FF
======================================================
FAIL: test_arithmetic (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 18, in test_arithmetic
    self.assertEqual(20, total_angle.degrees, 'Adding angles with
wrap-around')
AssertionError: 20 != 0 : Adding angles with wrap-around

======================================================
FAIL: test_degrees (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 7, in test_degrees
    self.assertEqual(60, small_angle.degrees)
AssertionError: 60 != 0

------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

Notice:

- Both test methods are shown. They both have a failed assertion highlighted.
- `test_degrees` makes several assertions, but only the first one has been run - once it fails, the others are not executed.
- For each failing assertion, you are given the line number; the expected and actual values; and its test method.
- The custom message in `test_arithmetic` shows up in the output.

This demonstrates one useful way to organize your test code. In a single test module (`test_angle.py`), you define one or more subclasses of `unittest.TestCase`. Here, I just define `TestAngle`, containing tests for the `Angle` class. Within this, I create several test methods, for testing different aspects of the class. And in each of these test methods, I can have as many assertions as makes sense.

Some of the naming conventions matter. It's traditional to start a test class name with the string `Test`, but that is not required; `unittest` will find all subclasses of `TestCase` automatically. But every method must start with the string "test". If it starts with anything else (even "Test"!), `unittest` will not run its assertions.

Running the test and watching it fail is an important first step. It verifies that the test does, in fact, actually test your code. As you write more and more tests, you'll occasionally create the test; run it, expecting it to fail; and find it

unexpectedly passes. That's a bug in your test code! Fortunately you ran the test first, so you caught it right away.

In the test code, we defined `test_degrees` before `test_arithmetic`, but they were actually run in the opposite order. It's important to craft your test methods to be self-contained, and not depend on one being run before the other; the order of execution is essentially random.[6]

At this point, we have a correctly failing test. If I'm using version control and working in a branch, this is a good commit point - check in the test code, because it specifies the correct behavior (even if it's presently failing). The next step is to actually make that test pass. Here's one way to do it:

```
# angle.py, version 2
class Angle:
    def __init__(self, degrees):
        self.degrees = degrees % 360
    def is_acute(self):
        return self.degrees < 90
    def add(self, other_angle):
        return Angle(self.degrees + other_angle.degrees)
```

Now when I run my test again, the test passes:

```
python3 -m unittest test_angle1.py
..
----------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This becomes your second commit in version control.

`assertEqual`, `assertTrue` and `assertFalse` will be the most common assertion methods you'll use, along with `assertNotEqual` (which does the opposite of `assertEqual`). Many others are provided, such as `assertIs`, `assertIsNone`, `assertIn`, and `assertIsInstance` - along with "not" variants (e.g. `assertIsNot`). Each takes a optional final message-string argument, like "Adding angles with wrap-around" in `test_arithmetic` above. If the test fails, this is printed in the output, which can give very helpful advice to whomever is troubleshooting a broken test.[7]

If you try checking that two dictionaries are equal, and they are not, the output is tailored to the data type: highlighting which key is missing, or which value is incorrect, for example. This also happens with lists, tuples, and sets, making troubleshooting much easier. What's actually happening is that `unittest` provides certain type-specialized assertions, like `assertDictEqual`, `assertListEqual`, and more. You almost never need to invoke them directly: if you invoke `assertEqual` with two dictionaries, it automatically dispatches to `assertDictEqual`, and similar for the other types. So you get this usefully detailed error reporting for free.

Notice the `assertEqual` lines take two arguments, and I always wrote the expected, correct value first:

```
small_angle = Angle(60)
self.assertEqual(60, small_angle.degrees)
```

It does not matter whether the expected value is first, or second. But it's smart to pick an order and stick with it - at least throughout a single

codebase, and maybe for all code you write. Sticking with a consistent order greatly improves the readability of your test output, because you never have to decipher which is which. Believe me, this will save you a lot of time in the long run. If you’re on a team, negotiate with them to agree on a consistent order.

Fixtures And Common Test Setup

As an application grows and you write more tests, you will find yourself writing groups of test methods that start or end with the same lines of code. This repeated code - which does some kind of pretest set-up, and/or after-test cleanup - can be consolidated in the special methods `setUp` and `tearDown`. When defined in your `TestCase` subclass, `setUp` is executed just before each test method starts; `tearDown` is run just after. This is repeated for every single test method.

Here's a realistic example of when you might use it. Imagine working on a tool that saves its state between runs in a special file, in JSON format. We'll call this the "state file". On start, it reads the state from the file; on exit, it rewrites it, if there are any changes. A stub of this class might look like

```
# statefile.py
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

In fleshing out this stub, we want our tests to verify the following:

- If I add a new key-value pair to the state, it is recorded correctly in the state file.
- If I alter the value of an existing key, that updated value is written to the state file.

- If the state is not changed, the state file's content stays the same.

For each test, we want the state file to be in a known starting state. Afterwards, we want to remove that file, so our tests don't leave garbage files on the filesystem. Here's how the `setUp` and `tearDown` methods accomplish this:

```
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'

class TestState(unittest.TestCase):
    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)

    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_remove_value(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
```

```
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(checked_content, INITIAL_STATE)
```

In `setUp`, we create a fresh temporary directory, and write the contents of `INITIAL_DATA` inside. Since we know each test will be working with a `State` object based on that initial data, we go ahead and create that object, and save it in `self.state`. Each test can then work with that object, confident it is in the same consistent starting state, regardless of what any other test method does. In effect, `setUp` creates a private sandbox for each test method.

The tests in `TestState` would all work reliably with just `setUp`. But we also want to clean up the temporary files we created; otherwise, they will accumulate over time with repeated test runs. The `tearDown` method will run after each `test_*` method completes, even if some of its assertions fail. This ensures the temp files and directories are all removed completely.

The generic term for this kind of pre-test preparation is called a *test fixture*. A test fixture is whatever needs to be done before a test can properly run. In this case, we set up the text fixture by creating the state file, and the `State` object. A text fixture can be a mock database; a set of files in a known state; some kind of network connection; or even starting a server process. You can do all these with `setUp`.

`tearDown` is for shutting down and cleaning up the text fixture: deleting files, stopping the server process, etc. For some kinds of tests, a tear-down might not be at all optional. If `setUp` starts some kind of server process, for example, and `tearDown` fails to terminate it, then `setUp` may not be able to run for the next test.

The camel-casing matters: people sometimes misspell them as `setup` or `teardown`, then wonder why they don't seem to be invoked. Also, any uncaught exception in either `setUp` or `tearDown` will cause `unittest` to mark the test method as failing (which means it will clearly show up in the test output), then immediately skip to the next test. For errors in `setUp`, this means none of that test's assertions will run (though it's still marked as failing). For `tearDown`, the test is marked as failing, even if all the individual assertions passed.

Asserting Exceptions

Sometimes your code is supposed to raise an exception, under certain exceptional conditions. If that condition occurs, and your code does *not* raise the correct exception, that's a bug. How do you write test code for this situation?

You can verify that behavior with a special method of `TestCase`, called `assertRaises`. It's used in a `with` statement in your test; the block under the `with` statement is asserted to raise the exception. For example, suppose you are writing a library that translates Roman numerals into integers. You might define a function called `roman2int`:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
```

In thinking about the best way to design this function, you decide that passing nonsensical input to `roman2int` should raise a `ValueError`. Here's how you write a test to assert that behavior:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

If you run this test, and `roman2int` does NOT raise the error, this is the result:

```
$ python3 -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("This is not a valid roman numeral.")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

When you fix the bug, and `roman2int` raises `ValueError` like it should, the test passes.

Using Subtests

Python 3 has a new feature called subtests, allowing you to conveniently iterate through a collection of test inputs. Imagine a function called `numwords`, which counts the number of unique words in a string (ignoring punctuation, spelling and spaces):

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Suppose you want to test how `numwords` handles excess whitespace. You can easily imagine a dozen different reasonable inputs that will result in the same return value, and want to verify it can handle them all. You might create something like this:

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

Seems a bit repetitive, doesn't it? The only thing varying is the argument to `numwords`. We might benefit from using a for loop:

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
```

```
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

At first glance, this is certainly more maintainable. If we add new variants, it's just another line in the `texts` list. And if I rename `numwords`, I only need to change it in one place in the test.

However, using a for loop like this creates more problems than it solves. Suppose you run this test, and get the following failure:

```
$ python3 -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in
test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Look closely, and you'll realize that `numwords` returned 3 when it was supposed to return 2. Pop quiz: out of all the inputs in the list, which caused the bad return value?

The way we've written the test, there is no way to know. All you can infer is that at least one of the test inputs produced an incorrect value. You don't know which one. That's the first problem. The second is that the test stops

when the first failure happens. If several test inputs are causing errors, it would be helpful to know that right away. (Of course, the original version has this shortcoming too.) Knowing all the failing inputs, and the incorrect results they create, would be *very* helpful for quickly understanding what's going on.

Python 3.4 introduced a new feature, called *subtests*, that gives you the best of all worlds. Our for-loop solution is actually quite close. All we have to do is add one line - can you spot it below?

```
    def test_whitespace_subtest(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

Just inside the for loop, we write `with self.subTest(text=text)`. This creates a context in which assertions can be made, and even fail. Regardless of whether they pass or not, the test continues with the next iteration of the for loop. At the end, *all* failures are collected and reported in the test result output, like this:

```
$ python3 -m unittest test_words_subtest.py

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(text='foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
```

```
test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(text='foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=2)
```

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

This is MUCH better. The two offending inputs are `"foo\tbar"` and `"foo bar \t \t"`. These are the only values containing tab characters, so you can quickly realize the nature of the bug: tab characters are being counted as separate words.

Let's look at the three key lines of code again:

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value arguments to `self.subTest` are used in reporting the output. They can be anything that helps you understand exactly what is wrong when a test fails. Often you will want to pass everything that varies from the test cases; here, that's only the string passed to `numwords`.

Be clear that in these three lines, the symbol `text` is used for two different things. The `text` variable in the for loop is the same variable that is passed to `numwords` on the last line. In the call to `subTest`, the left-hand side of `text=text` is actually a parameter that is used in the reporting output if the test fails. For example, suppose we wrote it as `input_text` instead:

```
        for text in texts:
            with self.subTest(input_text=text):
                self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords)
(input_text='foo\tbar')
```

In other words, the l-value `text` in the `assertEqual` line has nothing to do with the argument to `subTest`. Just remember that the arguments to `subTest` are only used in the error output when something goes wrong, and are otherwise ignored completely.

Final Thoughts

Let's recap the big ideas. Test-driven development means we create the test first, and whatever stubs we need to make the test run. We then run it, and watch it fail. **This is an important step.** You must run the test and see it fail.

This is important for two reasons. You don't really know if the test is correct until you verify that it **can** fail. As you write automated tests more and more over time, you will probably be surprised at how often you write a test, confident in its correctness, only to discover it passes when it should fail. As far as I can tell, every good, experienced software engineer I know still commonly experiences this… even after doing TDD for many years! This is why we build the habit of always verifying the test fails first.

The second reason is more subtle. As you gain experience with TDD and become comfortable with it, you will find the cycle of writing tests and making them pass lets you get into a state of flow. This means you are enjoyably productive and focused, in a way that is easy to maintain over time.

Is it important that you strictly follow test-driven development? People have different opinions on this, some of them *very* strong. Personally, I went through a period of almost a year where I followed TDD to the letter, very strictly. As a result, I got *very* good at writing tests, and writing high-quality tests very quickly.

Now that I've developed that level of skill, I prefer instead to follow the 80-20 rule, and sometimes the 70-30 or even 50-50 rule. I have noticed that TDD is most powerful when I have great clarity on the software's design,

architecture and APIs; it helps me get into an cognitive state that seems accelerated, so that I can more easily maintain my mental focus, and produce quality code faster.

But I find it very hard to write good tests when I don't yet have that clarity… when I am still thinking through how I will structure the program and organize the code. In fact, I find TDD slows me down in that phase, as any test I write will probably have to be completely rewritten several times, if not deleted, before things stabilize. In these situations, I prefer to get a first version of the code working through manual testing, then write the tests afterwards.

To close with the obvious: Experiment to find what approach works best for you, and not just follow what someone else writes that you "should" do. I encourage you to try TDD for a period of time, because of what it will teach you. But be flexible, and at some point step back and evaluate how you want to integrate it into your daily routine.

1 https://docs.python.org/3/library/unittest.html

2 You may be in a third category, having a lot of experience with a non-xUnit testing framework. If so, you should probably pretend you're in the first group. You'll be able to move quickly.

3 If you haven't done one of these yet, you will.

4 The object-oriented chapter talks about "magic methods" like `__add__`, which provide a more natural syntax for math-like operations on custom types. This chapter just uses regular methods, in case you haven't read that chapter yet.

5 Python 2 requires you to drop the test file's `.py` extension - in other words, passing the test module name. So you invoke it like `python2.7 -m unittest test_angles`. Python 3 lets you do either; we'll always use the test filename in this chapter, but you can use whichever you prefer.

6 If you find yourself wanting to run tests in a certain order, this might be better handled with `setUp` and `tearDown`, explained in the next section.

7 Which could be you, months or years down the road. Be considerate of your future self!

STRING FORMATTING

The situation with string formatting is complicated.

Once upon a time, Python introduced *percent formatting*. It uses `"%"` as a binary operator to render strings:

```
>>> drink = "coffee"
>>> price = 2.5
>>> message = "This %s costs $%.2f." % (drink, price)
>>> print(message)
This coffee costs $2.50.
```

Later in Python 2's history, a different style was introduced, called simply *string formatting* (yes, that's the official name). Its very different syntax makes any Python string a potential template, inserting values through the `str.format()` method.

```
>>> template = "This {} costs ${:.2f}."
>>> print(template.format(drink, price))
This coffee costs $2.50.
```

Python 3.6 introduces a third option, called *f-strings*. This lets you write literal strings, prefixed with an "f" character, interpolating values from the immediate context:

```
>>> message = f"This {drink} costs ${price:.02f}."
>>> print(message)
This coffee costs $2.50.
```

So… which do you use? Here's my guidance in a nutshell:

- Go ahead and master `str.format()` now. Everything you learn transfers entirely to f-strings, and you'll sometimes want to use `str.format()` even in cutting-edge versions of Python.
- Prefer f-strings when working in a codebase that supports it - meaning, all developers and end-users of the code base are certain to have Python 3.6 or later.
- Until then, prefer `str.format()`.
- Exception: for the `logging` module, use percent-formatting, even if you're otherwise using f-strings.
- Aside from `logging`, don't use percent-formatting unless legacy reasons force you to.

"Which should I use?" is a separate question from "which should a Python book use for its code examples?" As you've noticed, I am using `str.format()` throughout this book. That's because all modern Python versions support it, so I know everyone reading this book can use it.

Someday, when Python versions before 3.6 are a distant memory, there will be no reason not to use f-strings. But when that happens, `str.format()` will still be important. There are string formatting situations where f-strings are awkward at best, and `str.format()` is well suited. In the meantime,

there is a lot of Python code out there using `str.format()`, which you'll need to be able to read and understand. Hence, I'm choosing to focus on `str.format()` in this chapter. Conveniently, this also teaches you much about f-strings; they are more similar than different, because the formatting codes are nearly identical. `str.format()` is also the only practical choice for most people reading this, and will be for years still.

You might wonder if the old percent-formatting has any place in modern Python. In fact, it does, due to the `logging` module. As you'll read in its chapter, this important module is built on percent-formatting in a deep way. It's possible to use `str.format()` in new logging code, but requires special steps; and legacy logging code cannot be safely converted in an automated way. I recommend you just cooperate with the situation, and use percent-formatting for your log messages.

For those interested, this chapter ends with sections briefly explaining f-strings and percent-formatting. For now, we'll focus on `str.format()`. While reading, I highly recommend you have a Python interpreter prompt open, typing in the examples as you go along. The goal is to make its expressive power automatic and easy for you to use, so that it's *mentally* available to you… giving you the easy ability to use it in the future, without digging into the reference docs. Most people never master it to this threshold, effectively denying them most of the benefits of this rich tool. You won't have that problem.

Replacing Fields

`str.format()` lets you start simple, leveraging more complex extensions as needed. You start by creating a format string. This is just a regular string, and acts as a kind of template. It contains, among other text, one or more *replacement fields*. These are simply pairs of opening and closing curly braces: `"Good {}, my friend"`. You then invoke the `format` method on that string, passing in one argument for each replacement field:

```
>>> "Good {}, my friend".format("morning")
'Good morning, my friend'
>>> "Good {}, my friend".format("afternoon")
'Good afternoon, my friend'
>>> offer = "Give me {} dollars and I'll give you a {}."
>>> offer.format(2, "cheeseburger")
"Give me 2 dollars and I'll give you a cheeseburger."
>>> offer.format(7, "nice shoulder rub")
"Give me 7 dollars and I'll give you a nice shoulder rub."
```

(If possible, type these examples in an interpreter as you go along, so you learn them deeply.) `.format()` is a method returning a new string; the format string itself is not modified.

Notice how fields line up by position, and no type information is needed. The integer 2 and the string "cheeseburger" are both inserted without complaint. We'll see how to specify more precise types for the fields later.

Within the curly braces of the replacement field, you can specify numbers starting at 0. These reference the positions of the arguments passed to `format`, and allow you to repeat fields:

```
>>> "{0} is {1}; {1}, {0}".format("truth", "beauty")
'truth is beauty; beauty, truth'
```

You can also reference fields by a name, and pass the fields as key-value pairs to `format`:

```
>>> "Good {when}, {user}!".format(
         when="morning", user="John")
'Good morning, John!'
```

The arguments to `format()` don't actually have to be strings: they can be objects or lists. Reference within them as you normally would, within the curly braces:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> point = Point(3, 7)
>>> 'The coordinates are {point.x}, {point.y}'.format(
        point=point)
'The coordinates are 3, 7'
>>> 'The coordinates are {0.x}, {0.y}'.format(point)
'The coordinates are 3, 7'
```

Notice the difference in how you use the point with a named field (the first format call), versus a numbered field (the second). Here's how it looks with a list:

```
>>> params = ["morning", "user"]
>>> "Good {params[0]}, {params[1]}!".format(params=params)
'Good morning, user!'
>>> "Good {0[0]}, {0[1]}!".format(params)
'Good morning, user!'
```

You can do the same thing with dictionaries too, though there is one subtle quirk - can you spot it?

```
>>> params = {"when": "morning", "user":"John"}
>>> "Good {0[when]}, {0[user]}!".format(params)
'Good morning, John!'
```

See it? The key `when` in `{0[when]}` does not have quotation marks around it! In fact, if you do put them in, you get an error. This quirk was intentionally put in, to make it easier to reference keys within a string that is bounded by quotes already.

Number Formats (and "Format Specs")

Now that you know how to substitute values into different replacement fields (i.e., pairs of curly braces), you may next want to format a field as a number. Do this by inserting a colon between the curly braces, followed by one or more descriptive characters. For example, use ":d" to format as an integer, and ":f" to format as a floating-point number. Here's how it works:

```
>>> 'The magic number is {:d}'.format(42)
'The magic number is 42'
>>> 'The magic number is actually {:f}'.format(42)
'The magic number is actually 42.000000'
>>> 'But this will cause an error: {:d}'.format("foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'str'
```

The number 42 is rendered as either `42` or `42.000000`, depending on whether the replacement field is `{:d}` or `{:f}`. And when we try to stuff something that isn't a number in `{:d}`, it triggers a fatal error.

You can combine this with field numbering and naming. Just put that label before the colon:

```
>>> "The time is {hour:d} o' clock.".format(hour=11)
"The time is 11 o' clock."
>>> "The answer is {0:f}, not {1:f}. But you can round it to
{0:d}.".format(12, 14)
'The answer is 12.000000, not 14.000000. But you can round it to
12.'
```

The rule is easy: if you label a field - whether it's a string name, or a number - always put that label first within the curly brackets.

The part after after the colon is called a *format spec*. There's actually many options you can stuff in there, but let's focus on those related to formatting numbers first. As you saw, the code for an integer is `d`, converting the argument to an integer, if possible. If not, it throws a `ValueError`. (By the way, "d" stands for "decimal number", as in a base-ten number.)

Then there's floating point numbers. With the `f` code, we get six decimal places of precision by default - which are filled with zeros if necessary:

```
>>> from math import pi
>>> 'The ratio is about {:f}'.format(pi)
'The ratio is about 3.141593'
>>> '{:f} is NOT a good approximation.'.format(3)
'3.000000 is NOT a good approximation.'
```

"f" actually stands for "fixed-point number", not "floating point" - we'll see some variations later. We can change the number of fixed points: writing a period followed by a number means to use that many decimal places. We put it between the colon and the letter "f", like so:

```
>>> 'The ratio is about {:.3f}'.format(pi)
'The ratio is about 3.142'
>>> '{:.1f} is NOT a good approximation.'.format(3)
'3.0 is NOT a good approximation.'
```

It's easier for humans to read numbers with many digits if they have commas. You can tell the formatter to do this by putting a comma after the colon:

```
>>> "Billions and {:,d}'s".format(10**9)
"Billions and 1,000,000,000's"
>>> "It works with floating point too: {:,f}".format(10**9)
'It works with floating point too: 1,000,000,000.000000'
```

For large numbers, sometimes we want scientific notation, also called exponent notation. We can use the code "E" for that instead:

```
>>> "Billions and {:E}'s".format(10**9)
"Billions and 1.000000E+09's"
>>> "Precision works the same: {:.2E}".format(10**9)
'Precision works the same: 1.00E+09'
```

So far, we have seen codes for three presentation types: d (decimal) for integers; and f (fixed-point) and E (exponential) for floating-point numbers. We actually have many other choices for both: read the format-specification mini-language section[1] in the Python docs.

Width, Alignment, and Fill

In the examples above, the substituted values will take only as much space as they need, but no more. So `"a{}b".format(n)` will render as `a7b` if n is 7 - but not `a07b`, for example. But if n is 77, it will expand to take up four characters instead of three: `a77b`.

We can change this default behavior, placing the value in a field of a certain number of characters. If it's small enough to fit in there (i.e. not too many digits or chars), then it will be right-justified. We specify the width by putting the number of characters between the colon and the type code:

```
>>> "foo{:7d}bar".format(753)
'foo    753bar'
```

Let's count the character columns here:

```
foo    753bar
0123456789012
```

Positions 3 through 9 are taken up by the replacement field value. That value only has three characters (753), so the others are *filled* by the space character. We say that the space is the *fill* character here.

By default, the value is right-justified in the field for numbers. But for strings, it's left justified:

```
>>> "foo{:7s}bar".format("blah")
'fooblah   bar'
```

Generally speaking, it will default to right-justifying for any kind of number, and left-justify for everything else. We can override the default, or even just be explicit about what we want, by inserting an *alignment* right before the field width. For right-justifying, this is the greater-than sign:

```
>>> "foo{:>7d}bar".format(753)
'foo    753bar'
>>> "foo{:>7s}bar".format("blah")
'foo   blahbar'
```

To left-justify, use a less-than sign:

```
>>> "foo{:<7d}bar".format(753)
'foo753    bar'
>>> "foo{:<7s}bar".format("blah")
'fooblah   bar'
```

Or we can center it, with a caret:

```
>>> "foo{:^7d}bar".format(753)
'foo  753  bar'
>>> "foo{:^7s}bar".format("blah")
'foo blah  bar'
```

So far, the extra characters in the field have been spaces. That extra character is called the *fill character*, or the *fill*. We can specify a different fill character by placing it just before the alignment character (<, > or ^):

```
>>> "foo{:_>7d}bar".format(753)
'foo____753bar'
>>> "foo{:+<7d}bar".format(753)
'foo753++++bar'
>>> "foo{:X^7d}bar".format(753)
'fooXX753XXbar'
```

This is a good time to remember you can combine field names or indices with these format annotations - just put the name or index to the left side of the colon:

```
>>> "alpha{x:_<6d}beta{y:+^7d}gamma".format(x=42, y=17)
'alpha42____beta++17+++gamma'
>>> "{0:_>6d}{1:-^7d}{0:_<6d}".format(11, 333)
'____11--333--11____'
```

Historically, over the long and ongoing lifetime of `printf`, it's been common to use zero as a fill character for right-justified integer fields. So if the number 42 is put in a five-character-wide field, it shows up at "00042". Since people often want to do this, a shorthand evolved. You can omit the alignment character if the fill is "0" (zero), and the type is decimal:

```
>>> "foo{:0>7d}bar".format(753)
'foo0000753bar'
>>> "foo{:07d}bar".format(753)
'foo0000753bar'
```

That doesn't generally work for other fill values, though:

```
>>> "foo{:_7d}bar".format(753)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Invalid format specifier
```

What happens if the value is too big to fit in a tiny field? The width is actually a *minimum* width. So it will expand as needed to fill it out:

```
>>> "red{:4d}green".format(123456789)
'red123456789green'
```

There isn’t any way to specify a maximum width. If you need that, you can convert it to a string, implement your own trimming logic, then inject that trimmed string:

```
>>> value = 123456789 # Or some other number.
>>> trimmed_value = str(value)[:4] # Or last 4, etc.
>>> "red{:>4s}green".format(trimmed_value)
'red1234green'
```

F-Strings

Python 3.6 introduced an alternative to `str.format()`, called **f-strings**. The formal name is "formatted string literal". Instead of a `.format()` method, you prefix the string with the letters "f" or "F", putting variable names directly inside the replacement field:

```
>>> time_of_day = "afternoon"
>>> f"Good {time_of_day}, my friend"
'Good afternoon, my friend'
>>> F"Good {time_of_day}, my friend"
'Good afternoon, my friend'
```

It's exactly equivalent to this:

```
>>> # This...
... "Good {time_of_day}, my friend".format(
...     time_of_day=time_of_day)
'Good afternoon, my friend'
>>> # Or this:
... "Good {}, my friend".format(time_of_day)
'Good afternoon, my friend'
```

If your values are already stored in locally-readable variables, using f-strings is more succinct. But you can do more than that. In fact, the replacement field (i.e. the curly braces) can contain not only a variable name, but a full Python expression!

```
>>> f"Good {time_of_day.upper()}, my friend"
'Good AFTERNOON, my friend'

>>> def reverse(string):
...     return string[::-1]
>>> f"Good {reverse(time_of_day)}, my friend"
'Good noonretfa, my friend'
```

```
>>> groceries = ["milk", "bread", "broccoli"]
>>> f"I need to get some {groceries[2]}."
'I need to get some broccoli.'
```

You can do this with `str.format()`, too, but f-strings express it a bit more naturally. Aside from that, you can use the normal number formatting codes. After the expression name, simply write a colon, and the same code you would use for `str.format()`:

```
>>> from math import pi
>>> f"The ratio is about {pi:f}"
'The ratio is about 3.141593'
>>> f"Which is roughly {pi:0.2f}"
'Which is roughly 3.14'
>>> f"{pi:.0f} is NOT a good approximation."
'3 is NOT a good approximation.'

>>> number = 10**9
>>> f"Billions and {number:,d}'s"
"Billions and 1,000,000,000's"
>>> f"Billions and {number:E}'s"
"Billions and 1.000000E+09's"
```

As well as the width, alignment, and fill:

```
>>> num = 753
>>> word = "WOW"
>>> f"foo{num:7d}bar"
'foo    753bar'
>>> f"foo{word:7s}bar"
'fooWOW    bar'
>>> f"foo{word:>7s}bar"
'foo    WOWbar'
>>> f"foo{num:<7d}bar"
'foo753    bar'
>>> f"foo{num:^7d}bar"
```

```
'foo  753  bar'
>>> f"foo{num:X^7d}bar"
'fooXX753XXbar'
```

Now it's clear why I emphasize `str.format()` in this chapter, and this book. Practically speaking, all Python programmers need to know `str.format()` anyway; and once you've learned it, you are fluent with f-strings almost immediately.

The main downside to f-strings will become less important over time. It only works with Python 3.6 and later. That means in order to use f-strings in your code, you must be working on a codebase which will only ever be executed on those versions. This applies not only to your fellow developers, but - more problematically - all end users. If a customer has installed Python 3.5 on a server, and your program uses f-strings, they won't be able to run it unless you can convince them to upgrade. Which, unfortunately, probably isn't as high a priority for them as it is for you.

As I write this, f-strings have been out a short while. But many Python developers seem to have already fallen in love with them. The next edition of this book may emphasize them more, depending how quickly the Python community moves to versions of Python which support f-strings, and how popular they become. Fortunately, it's quite easy to switch back and forth between `str.format()` and f-strings; there seems to be very little mental energy needed to switch. So if you want to use f-strings, you can do so when it's practical, and then easily switch to `str.format()` when needed.

Percent Formatting

Modern Python still needs percent formatting in a few places, mainly when you work with the `logging` module. Thankfully, you don't need to know all its details. Learning just a few parts of percent formatting will cover 95% of what you're likely to need. I'll focus on that high-impact portion here; for more detail, consult the official Python reference.[2]

Percent formatting is officially called "printf-style string formatting", and - like the string formatting in *many* languages - has its roots in C's `printf()`. So if you have experience with that, you'll skim through quickly - though there *are* a few differences.

It uses the percent character in two ways. Here's a simple example:

```
>>> "Hello, %s, today is %s." % ("Aaron", "Tuesday")
'Hello, Aaron, today is Tuesday.'
```

Notice percent is used as a binary operator. On its left is a string; on its right, a tuple of two strings. That string on the left is called the *format*; you can see it has two percent characters inside. Specifically, it has the sequence `%s` twice. These `%s` sequences are called *conversion specifiers*. There's two of them, and two values in the tuple on the right; they map to each other. Each value is substituted for its corresponding `%s`.

The "s" means that the inserted value is converted to a string; these are already strings, though, so they are just placed in. You can also use `%d` for an integer:

```
>>> "Here's %d dollars and %d cents." % (14, 25)
"Here's 14 dollars and 25 cents."
```

Sometimes it doesn't matter much which specifier you use. If that last format string was `"Here's %s dollars and %s cents."`, it would render the same. But I recommend you choose the strictest type that will work for your data; if you expect the value to be an integer, use `%d`, so that if it's *not* an integer, you'll get a clear stack trace telling you what's wrong:

```
>>> "Here's %d dollars and %d cents." % (14, "a quarter")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
```

(I'm assuming you'd rather discover a lurking bug now, during development, instead of through an angry customer's bug report later.)

Sometimes you'll have just one value to interject. A tuple of one value must be written with an extra comma, like `(foo,)` - because `(foo)` becomes simply `foo` in the normal meaning of parentheses for grouping. So to format a string of one variable, you can type this:

```
>>> "High %d!" % (5,)
'High 5!'
```

Early in Python's history, people decided typing those extra characters was a bit annoying, especially given how common it is to format a string with a single value. So as a special case, when you have a single specifier, you can pass in a single value instead of a tuple:

```
>>> "High %d!" % 5
'High 5!'
```

In addition to `%d` and `%s`, you can use `%f` for a floating-point number:

```
>>> "I owe you $%f." % 7.05
'I owe you $7.050000.'
```

You'll often want to specify its precision - the number of digits to the right of the decimal. Do this by inserting a *precision code* between `%` and `f` - which will be a `.` (dot) followed by an integer:

```
>>> "I owe you $%.2f." % 5.05
'I owe you $5.05.'
```

You can also use `%r` (which formats the `repr()` of the object). It's especially useful for logging and troubleshooting:

```
>>> class Money:
...     def __init__(self, dollars, cents):
...         self.dollars = dollars
...         self.cents = cents
...     def __repr__(self):
...         return 'Money({},{})'.format(
...             self.dollars, self.cents)
>>> cash = Money(127, 82)
>>> "Cash on hand: %r" % cash
'Cash on hand: Money(127,82)'
```

There is much more to percent formatting than this, but what we've covered lets you read and write most of what you need.

Now, as mentioned, in modern Python you'll mainly need to use percent formatting with the `logging` module. However, in that case, you use it a bit differently. As described in its chapter, the `logging` module includes

functions for log events at different levels of urgency - whether that's an error, a warning, or even a non-urgent informational message:

```
logging.info("So far, so good!")
```

You will *very* often want to inject run-time values into the message. For example, if a customer spends a certain amount of money:

```
logging.info("User %s spent $%0.2f", username, amount)
```

Notice there's no binary percent operator! That's deliberate. The logging functions are designed to take a format string as the first argument, and the values to substitute as subsequent arguments. That's because not every log message needs to be executed. You can - and often will - configured your logger to filter out all those boring `info` messages, for example, or omit the overly detailed `debug` messages. In other words, don't do this:

```
# NO! Bad code!
logging.info("User %s spent $%0.2f" % (username, amount))
```

Suppose you are filtering out `info` messages right now, so the `info` message doesn't need to actually log. In the first, recommended form, that `logging.info` line in your code is cheap; it's essentially treated by Python as a no-op. In the second form, it will still be translated as a no-op, *but only after that string is rendered.* So you unnecessarily incur the cost of rendering the string, just to throw it away. This is all explained in more detail in the logging chapter; for now, just be aware of the idea.

1 https://docs.python.org/3/library/string.html#format-specification-mini-language

2 https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

LOGGING IN PYTHON

Logging is critical in many kinds of software. For long-running software systems, it enables continuous telemetry and reporting. And for *all* software, it can provides priceless information for troubleshooting and post-mortems. The bigger the application, the more important logging becomes. But even small scripts can benefit.

Python provides logging through the `logging` module. In my opinion, this module is one of the more technically impressive parts of Python's standard library. It's well-designed, flexible, thread-safe, and richly powerful. It's also complex, with many moving parts, making it hard to learn well. This chapter gets you over most of that learning curve, so you can fully benefit from what `logging` has to offer. The payoff is well worth it, and will serve you for years.

Broadly, there are two ways to use `logging`. One, which I'm calling the *basic interface*, is appropriate for scripts - meaning, Python programs that are small enough to fit in a single file. For more substantial applications, it's typically better to use *logger objects*, which give more flexible, centralized control, and access to logging hierarchies. We'll start with the former, to introduce key ideas.

The Basic Interface

Here's the easiest way to use Python's logging module:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed to standard output:

```
WARNING:root:Look out!
```

You can do useful logging right away, by calling functions in the `logging` module itself. Notice you invoke `logging.warning()`, and the output line starts with `WARNING`. You can also call `logging.error()`, which gives a different prefix:

```
ERROR:root:Look out!
```

We say that `warning` and `error` are at different *message log levels*. You have a spectrum of log levels to choose from, in order of increasing severity: [1]

debug

Detailed information, typically of interest only when diagnosing problems.

info

Confirmation that things are working as expected.

warning

An indication that something unexpected happened, or indicative of some

problem in the near future (e.g. â€˜disk space lowâ€™). The software is still working as expected.

error

Due to a more serious problem, the software has not been able to perform some function.

critical

A serious error, indicating that the program itself may be unable to continue running.

You use them all just like `logging.warning()` and `logging.error()`:

```
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

Each has a corresponding uppercased constant in the library (e.g., `logging.WARNING` for `logging.warning()`). You use these when defining the *log level threshold*. Run the above, and here is the output:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

Where did the debug and info messages go? As it turns out, the default logging threshold is `logging.WARNING`, which means only messages of that severity or greater are actually generated; the others are ignored

completely. The order matters in the list above; `debug` is considered strictly less severe than `info`, and so on. Change the log level threshold using the `basicConfig` function:

```
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Again, the order is `debug()`, `info()`, `warning()`, `error()` and `critical()`, from lowest to highest severity. When we set the log level threshold, we declare that we only want to see messages of that level or higher. Messages of a lower level are not printed. When you set `level` to `logging.DEBUG`, you see everything; set it to `logging.CRITICAL`, and you only see critical messages, and so on.

The phrase "log level" means two different things, depending on context. It can mean the severity of a message, which you set by choosing which of the functions to use - `logging.warning()`, etc. Or it can mean the threshold for ignoring messages, which is signaled by the constants: `logging.WARNING`, etc.

You can also use the constants in the more general `logging.log` function - for example, a debug message:

```
logging.log(logging.DEBUG,
    "Small detail. Useful for troubleshooting.")
logging.log(logging.INFO, "This is informative.")
logging.log(logging.WARNING, "This is a warning message.")
logging.log(logging.ERROR, "Uh oh. Something went wrong.")
logging.log(logging.CRITICAL, "We have a big problem!")
```

This lets you modify the log level dynamically, at runtime:

```
def log_results(message, level=logging.INFO):
    logging.log(level, "Results: " + message)
```

Why do we have log levels?

If you haven't worked with similar logging systems before, you may wonder why we have different log levels, and why you'd want to control the filtering threshold. It's easiest to see this if you've written Python scripts that include a number of `print()` statements - including some useful for diagnosis when something goes wrong, but a distraction when everything is working fine.

The fact is, some of those `print()` statements are more important than others. Some indicate mission-critical problems you always want to know about - possibly to the point of waking up an engineer, so they can deploy a fix immediately. Some are important, but can wait until the next work day - and you definitely do NOT want to wake anyone up for that. Some are details which may have been important in the past, and might be in the future, so you don't want to remove them; in the meantime, they are just line noise.

Having log levels solves all these problems. As you develop and evolve your code over time, you continually add new logging statements of the

appropriate severity. You now even have the freedom to be proactive. With "logging" via `print()`, each log statement has a cost - certainly in signal-to-noise ratio, and also potentially in performance. So you might debate whether to include that print statement at all. But with `logging`, you can insert `info` messages, for example, to log certain events occurring as they should. In development, those INFO messages can be very useful to verify certain things are happening, so you can modify the log level to produce them. On production, you may not want to have them cluttering up the logs, so you just set the threshold higher. Or if you are doing some kind of monitoring on production, and temporarily need that information, you can adjust the log level threshold to output those messages; when you are finished, you can adjust it back to exclude them again.

When troubleshooting, you can liberally introduce `debug`-level statements to provide extra detailed statements. When done, you can just adjust the log level to turn them off. You can leave them in the code without cost, eliminating any risk of introducing more bugs when you go through and remove them. This also leaves them available if they are needed in the future.

The log level symbols are actually set to integers. You can theoretically use these numbers instead, or even define your own log levels that are (for example) a third of the way between `WARNING` and `ERROR`. In normal practice, it’s best to use the predefined logging levels. Doing otherwise makes your code harder to read and maintain, and isn’t worthwhile unless you have a compelling reason.

For reference, the numbers are 50 for `CRITICAL`, 40 for `ERROR`, 30 for `WARNING`, 20 for `INFO`, and 10 for `DEBUG`. So when you set the log level threshold, it's actually setting a number. The only log messages emitted are those with a level greater than or equal to that number.

Configuring The Basic Interface

You saw above you can change the loglevel threshold by calling a function called `basicConfig`:

```
logging.basicConfig(level=logging.INFO)
logging.debug("You won't see this message!")
logging.error("But you will see this one.")
```

If you use it at all, `basicConfig` must be called exactly once, and it must happen before the first logging event. (Meaning, before the first call to `debug()`, or `warning()`, etc.) Additionally, if your program has several threads, it must be called from the main thread - and *only* the main thread. [2]

You already met one of the configuration options, `level`. This is set to the log level threshold, and is one of `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`. Some of the other options include:

`filename`

Write log messages to the given file, rather than stderr.

`filemode`

Set to "a" to append to the log file (the default), or "w" to overwrite.

`format`

The format of log records.

`level`

The log level threshold, described above.

By default, log messages are written to standard error. You can also write them to a file, one per line, to easily read later. Do this by setting `filename` to the log file path. By default it appends log messages, meaning that it will only add to the end of the file if it isn't empty. If you'd rather the file be emptied before the first log message, set `filemode` to "w". Be careful about doing that, of course, because you can easily lose old log messages if the application restarts:

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

The other valid value is "a", for append - that's the default, and probably will serve you better in production. "w" can be useful during development, though.

`format` defines what chunks of information the final log record will include, and how they are laid out. These chunks are called "attributes" in the `logging` module docs. One of these attributes is the actual log message - the string you pass when you call `logging.warning()`, and so on. Often you will want to include other attributes as well. Consider the kind of log record we saw above:

```
WARNING:root:Collision imminent
```

This record has three attributes, separated by colons. First is the log level name; last is the actual string message you pass when you call `logging.warning()`. (In the middle is the name of the underlying logger

object. `basicConfig` uses a logger called "root"; we'll learn more about that later.)

You specify the layout you want by setting `format` to a string that defines certain named fields, according to percent-style formatting. Three of them are `levelname`, the log level; `message`, the message string passed to the logging function; and `name`, the name of the underlying logger. Here's an example:

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

If you run this as a program, you get the following output:

```
Log level: WARNING, msg: Collision imminent
```

It turns out the default formatting string is

```
%(levelname)s:%(name)s:%(message)s
```

You indicate named fields in percent-formatting by `%(FIELDNAME)X`, where "X" is a type code: `s` for string, `d` for integer (decimal), and `f` for floating-point.

Many other attributes are provided, if you want to include them. Here's a select few from the full list:[3]

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time

funcName	`%(funcName)s`	Name of function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
name	`%(name)s`	The logger's name

You might be wondering why log record format strings use Python 2's percent-formatting style, when everything else in Python 3 uses the newer, brace-style string formatting. As it turns out, the conversion was attempted, but backwards-compatibility reasons made percent-formatting the only practical choice for the logging module, even after the Python 3 reboot.

If you want to use the newer string formatting badly enough, there are things you can do - there's even a standard recipe.[4] But doing so is complicated enough that it may not be worth the effort, and it won't help with legacy code. I recommend you simply cooperate with the situation, and use percent formatting with your Python logging.

Passing Arguments

You often want to include some kind of runtime data in the logged message. When you construct the message to log, specify the final log message like this:

```
num_fruits = 14
fruit_name = "oranges"
logging.info(
    "We ate %d of your %s. Thanks!",
    num_fruits, fruit_name)
```

The output:

```
INFO:root:We ate 14 of your oranges. Thanks!
```

We call `info` with three parameters. First is the format string; the second and third are arguments. The general form is

```
logging.info(format, *args)
```

You can pass zero or more arguments, so long as each has a field in the format string:

```
logging.info("%s, %s, %s, %s, %s, %s and %s",
    "Doc", "Happy", "Sneezy", "Bashful",
    "Dopey", "Sleepy", "Grumpy")
```

You *must* resist the obvious temptation to format the string fully, and pass that to the logging function:

```
num_fruits = 14
fruit_name = "oranges"
logging.warning(
```

```
    "Don't do this: %d %s" % (num_fruits, fruit_name))
logging.warning(
    "Or even this: {:d} {:s}".format(
    num_fruits, fruit_name))
```

This works, of course, in the sense that you will get correct log messages. However, it's wasteful, and surrenders important benefits `logging` normally provides. Remember: when the line of code with the log message is executed, it may not actually trigger a log event. If the log level threshold is higher than the message itself, the line does nothing. In that case, there is no reason to format the string.

In the first form, the string is formatted if and only if a log event actually happens, so that's fine. But if you format the string yourself, it's *always* formatted. That takes up system memory and CPU cycles even if no logging takes place. If the code path with the logging call is only executed occasionally, that's not a big deal. But it impairs the program when a debug message is logged in the middle of a tight loop. When you originally code the line, you never really know where it might migrate in the future, or who will call your function in ways you never imagined.

So just use the supported form, where the first argument is the format string, and subsequent arguments are the parameters for it. You can also use named fields, by passing a dictionary as the second argument:

```
fruit_info = {"count": 14, "name": "oranges"}
logging.info(
    "We ate %(count)d of your %(name)s. Thanks!",
    fruit_info)
```

Beyond Basic: Loggers

The basic interface is simple and easy to set up. It works well in single-file scripts. Larger Python applications tend to have different logging needs, however. `logging` meets these needs through a richer interface, called *logger objects* - or simply, *loggers*.

Actually, you have been using a logger object all along: when you call `logging.warning()` (or the other log functions), they convey messages through what is called the *root logger* - the primary, default logger object. This is why the word "root" shows in some example output.

`logger.basicConfig` operates on this root logger. You can fetch the actual root logger object by calling `logging.getLogger`:

```
>>> logger = logging.getLogger()
>>> logger.name
'root'
```

As you can see, it knows its name is "root". Logger objects have all the same functions (methods, actually) the `logging` module itself has:

```
import logging
logger = logging.getLogger()
logger.debug("Small detail. Useful for troubleshooting.")
logger.info("This is informative.")
logger.warning("This is a warning message.")
logger.error("Uh oh. Something went wrong.")
logger.critical("We have a big problem!")
```

Save this in a file and run it, and you'll see the following output:

```
This is a warning message.
Uh oh. Something went wrong.
We have a big problem!
```

This is different from what we saw with `basicConfig`, which printed out this instead:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

At this point, we've taken steps backward compared to `basicConfig`. Not only is the log message unadorned by the log level, or anything else useful. The log level threshold is hard-coded to `logging.WARNING`, with no way to change it. The logging output will be written to standard error, and no where else, regardless of where you actually need it to go.

Let's take inventory of what we want to control here. We want to choose our log record format. And further, we want to be able to control the log level threshold, and write messages to different streams and destinations. You do this with a tool called *handlers*.

Log Destinations: Handlers and Streams

By default, loggers write to standard error. You can select a different destination - or even *several* destinations - for each log record:

- You can write log records to a file. Very common.
- You can, while writing records to that file, *also* parrot it to stderr.
- Or to stdout. Or both.
- You can simultaneously log messages to two different files.
- In fact, you can log (say) `INFO` and higher messages to one file, and `ERROR` and higher to another.
- You can write log records to a remote log server, accessed via a REST HTTP API.
- Mix and match all the above, and more.
- And you can set a different, custom log format for each destination.

This is all managed through what are called *handlers*. In Python logging, a handler's job is to take a log record, and make sure it gets recorded in the appropriate destination. That destination can be a file; a stream like stderr or stdout; or something more abstract, like inserting into a queue, or transmitting via an RPC or HTTP call.

By default, logger objects don't have any handlers. You can verify this using the `hasHandlers` method:

```
>>> logger = logging.getLogger()
>>> logger.hasHandlers()
False
```

With no handler, a logger has the following behavior:

- Messages are written to stderr.
- Only the message is written, nothing else. There's no way to add fields or otherwise modify it.
- The log level threshold is `logging.WARNING`. There is no way to change that.

To change this, your first step is to create a handler. Nearly all logger objects you ever use will have custom handlers. Let's see how to create a simple handler that writes messages to a file, called "log.txt".

```
import logging
logger = logging.getLogger()
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

The `logging` module provides a class called `FileHandler`. It takes a file path argument, and will write log records into that file, one per line. When you run this code, `log.txt` will be created (if it doesn't already exist), and will contain the string "Boo!" followed by a newline. (If `log.txt` did exist already, the logged message would be *appended* to the end of the file.)

But "A little detail" is not written, because it's below the default logger threshold of `WARNING`. We change that by calling a method named `setLevel` on the logger object:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

This writes the following in "log.txt":

```
A little detail
Boo!
```

Confusingly, you can call `setLevel` on a logger with no handlers, *but it has no effect*:

```
# Doing it wrong:
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) # No effect.
logger.debug("This won't work :(")
```

To change the threshold from the default of `logging.WARNING`, you must both add a handler, *and* change the logger's level.

What if you want to log to stdout? Do that with a `StreamHandler`:

```
import logging
import sys
logger = logging.getLogger()
```

```
out_handler = logging.StreamHandler(sys.stdout)
logger.addHandler(out_handler)
logger.warning("Boo!")
```

If you save this in a file and run it, you'll get "Boo!" on standard output. Notice that `logging.StreamHandler` takes `sys.stdout` as its argument. You can create a `StreamHandler` without an argument too, in which case it will write its records to standard error:

```
import logging
logger = logging.getLogger()
# Same as StreamHandler(sys.stderr)
stderr_handler = logging.StreamHandler()
logger.addHandler(stderr_handler)
logger.warning("This goes to standard error")
```

In fact, you can pass any file-like object; The object just needs to define compatible `write` and `flush` methods. Theoretically, you could even log to a file by creating a handler like `StreamHandler(open("log.txt", "a"))` - but in that case, it's better to use a `FileHandler`, so it can manage opening and closing the file.

When creating a handler, your needs are nearly always met by either `StreamHandler` or `FileHandler`. There are other predefined handlers, too, useful when logging to certain specialized destinations:

- `WatchedFileHandler` and `RotatingFileHandler`, for logging to rotated log files
- `SocketHandler` and `DatagramHandler` for logging over network sockets

- `HTTPHandler` for logging over an HTTP REST interface
- `QueueHandler` and `QueueListener` for queuing log records across thread and process boundaries

See the official docs[5] for more details.

Logging to Multiple Destinations

Suppose you want your long-running application to log all messages to a file, including debug-level records. At the same time, you want warnings, errors, and criticals logged to the console. How do you do this?

We've given you part of the answer already. A single logger object can have multiple handlers: all you have to do is call `addHandler` multiple times, passing a different handler object for each. For example, here is how you parrot all log messages to the console (via standard error) and also to a file:

```
import logging
logger = logging.getLogger()
# Remember, StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning(
    "This goes to both the console, AND log.txt.")
```

This is combining what we learned above. We create two handlers - a `StreamHandler` named `console_handler`, and a `FileHandler` named `logfile_handler` - and add both to the same logger (via `addHandler`). That's all you need to log to multiple destinations in parallel. Sure enough, if you save the above in a script and run it, you'll find the messages are both written into "log.txt", as well as printed on the console (through standard error).

We aren't done, though. How do we make it so every record is written in the log file, but only those of `logging.WARNING` or higher get sent to the console screen? Do this by setting log level thresholds for both the logger object and the individual handlers. Both logger objects and handlers have a method called `setLevel`, taking a log level threshold as an argument:

```
my_logger.setLevel(logging.DEBUG)
my_handler.setLevel(logging.INFO)
```

If you set the level for a logger, but not its handlers, the handlers inherit from the logger:

```
my_logger.setLevel(logging.ERROR)
my_logger.addHandler(my_handler)
my_logger.error("This message is emitted by my_handler.")
my_logger.debug("But this message will not.")
```

And you can override that at the handler level. Here, I create two handlers. One handler inherits its threshold from the logger, while the other does its own thing:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

verbose_handler = logging.FileHandler("verbose.txt")
logger.addHandler(verbose_handler)

terse_handler = logging.FileHandler("terse.txt")
terse_handler.setLevel(logging.WARNING)
logger.addHandler(terse_handler)

logger.debug("This message appears in verbose.txt ONLY.")
logger.warning("And this message appears in both files.")
```

There's a caveat, though: a handler can only make itself *more* selective than its logger, not less. If the logger chooses a threshold of `logger.DEBUG`, its handler can choose a threshold of `logger.INFO`, or `logger.ERROR`, and so on. But if the logger defines a strict threshold - say, `logger.INFO` - an individual handler cannot choose a lower one, like `logger.DEBUG`. So something like this won't work:

```
# This doesn't quite work...
import logging
my_logger = logging.getLogger()
my_logger.setLevel(logging.INFO)
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
my_logger.debug("No one will ever see this message :(")
```

There's a subtle corollary of this. By default, a logger object's threshold is set to `logger.WARNING`. So if you don't set the logger object's log level at all, it implicitly censors all handlers:

```
import logging
my_logger = logging.getLogger()
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
# No one will see this message either.
my_logger.debug(
    "Because anything under WARNING gets filtered.")
```

The logger object's default log level is not always permissive enough for all handlers you might want to define. So you will generally want to start by setting the logger object to the lowest threshold needed by any log-record destination, and tighten that threshold for each handler as needed.

Bringing this all together, we can now accomplish what we originally wanted - to verbosely log everything into a log file, while duplicating only the more interesting messages onto the console:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
# Warnings and higher only on the console.
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)
logger.addHandler(console_handler)
# But allow everything to into the log file.
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)

logger.warning(
    "This goes to both the console, AND into log.txt.")
logger.debug("While this only goes to the file.")
```

Add as many handlers as you want. Each can have different log levels. You can log to many different destinations, using the different built-in handler types mentioned above. If those don't do what you need, implement your own subclass of `logging.Handler` and use that.

Record Layout with Formatters

We haven't covered one important detail. So far, we've only shown you how to create logger objects that will write just the log message and nothing else. At the very least, you probably want to annotate that with the log level. You may also want to insert the time, or some other information. How do you do that?

The answer is to use a *formatter*. A formatter converts a log record into something that is recorded in the handler's destination. That's an abstract way of saying it; more simply, a typical formatter just converts the record into a usefully-formatted string. That string contains the actual log message, as well as the other fields you care about.

The procedure is to create a `Formatter` object, then associate with a handler (using the latter's `setHandler` method). Creating a formatter is easy - it normally takes just one argument, the format string:

```
import logging
my_handler = logging.StreamHandler()
fmt = logging.Formatter("My message is: %(message)s")
my_handler.setFormatter(fmt)
my_logger = logging.getLogger()
my_logger.addHandler(my_handler)
my_logger.warning("WAKE UP!!")
```

If you run this in a script, the output will be:

```
My message is this: WAKE UP!!
```

Notice the attribute for the message, `%(message)s`, included in the string. This is just a normal formatting string, in the older, percent-formatting style. It's exactly equivalent to using the `format` argument when you call `basicConfig`. For this reason, you can use the same attributes, arranged however you like - here's the attribute table again, distilled from the full official list:[6]

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Name of function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
name	`%(name)s`	The logger's name

1 These beautifully crisp descriptions, which I cannot improve upon, are taken from https://docs.python.org/3/howto/logging.html .

2 These restrictions aren't in place for logger objects, described later.

3 https://docs.python.org/3/library/logging.html#logrecord-attributes

4 https://docs.python.org/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles

5 https://docs.python.org/3/library/logging.handlers.html

6 https://docs.python.org/3/library/logging.html#logrecord-attributes

WHAT'S NEXT?

How lucky we are. Our craft of software is profoundly rewarding - personally, professionally, financially… and in terms of the massive, positive impact we can have. It's deeply fulfilling, in ways those who don't code may never know.

On top of that, we have a wonderful language like Python to code in. When I first met Guido van Rossum in person, I thanked him for creating my favorite language. If you ever meet him, do the same!

This is not an end. It's a starting point. And I mean that: I already have a detailed outline of this book's 3rd edition, and even some of the *fourth*. And at least *two* other completely different books mapped out, about Python or the broader craft of software development. *And* other media like videos, *and* live, in-person events. All I will say is: keep an eye on powerfulpython.com.

So… we're not done. Maybe you found the hidden message earlier: "This book is just the beginning. Far more to come." And while I can't yet reveal what that fully means, I hope you're just as excited as I for the future.

My email is aaron@powerfulpython.com. I know there are topics you have always wanted to learn about, for Python or programming in general. As you

think of them, tell me; your requests deeply influence what I create next. Even if I don't reply (sometimes my inbox just gets buried), I always carefully read and consider what you have to say.

Thank you for reading; it is my honor writing for you. My prayer is that you will find it valuable in your work, your career, even - dare I hope? - your life. If we ever cross paths, please introduce yourself. I'd love to meet you.

Happy coding!

Aaron Maxwell